فصل اول

معرفی کامپیوتر، اینترنت و وب

اهداف

- درک مفاهیم سخت‌افزاری و نرم‌افزاری.
- درک مفاهیم اولیه تکنولوژی شی، همانند کلاس‌ها، شی‌ها، صفات، رفتار، کپسوله‌سازی و توارث.
- آشنایی با انواع زبان‌های برنامه‌نویسی.
- آشنایی با زبان‌های برنامه‌نویسی پرکاربرد.
- آشنایی با محیط توسعه برنامه C++.
- تاریخچه زبان مدل‌سازی UML.
- تاریخچه اینترنت و وب.
- استفاده از عملگرهای محاسباتی.
- تست برنامه‌های C++ بر روی لینوکس و ویندوز XP.

رئوس مطالب

۱-۱ مقدمه

به ++C خوش آمدید! ما خیلی سخت تلاش کرده‌ایم تا کتابی به رشته تحریر در آوریم که حاوی اطلاعات مفیدی بوده و سرگرم کننده نیز باشد و شما را وادار به یادگیری کند. اصولا زبان ++C زبانی است که اکثر مخاطبان آن برنامه نویسان با تجربه هستند، از اینرو سعی کرده‌ایم تا این کتاب برای افراد زیر مناسب باشد:

- افرادی که تا حدی دارای تجربه برنامه‌نویسی بوده یا فاقد تجربه هستند.
- افرادی که دارای تجربه برنامه‌نویسی بوده و مایل هستند تا درک عمیقتری نسبت به این زبان پیدا کنند.

نقطه قوت این کتاب وضوح و ترتیب خاص بیان انواع تکنیک‌های برنامه‌نویسی همانند برنامه‌نویسی ساخت‌یافته، و شی‌گرا (OOP) است. هیچ برنامه‌نویسی، برنامه‌نویسی را بخوبی یاد نخواهد گرفت مگر آنکه از ابتدا در مسیر صحیح قرار بگیرد. ما سعی کرده‌ایم که بطور واضح و خیلی سرراست به موضوعات نزدیک شویم. این کتاب حاوی مثال‌های فراوانی است که در یادگیری بسیار موثر واقع می‌شوند. در این کتاب

خروجی تمام برنامه‌ها آورده شده است. تمام برنامه‌های معرفی شده در این کتاب بر روی CD همراه کتاب موجود می باشند. فصل‌های اولیه در ارتباط با مفاهیم بنیادین کامپیوترها، برنامه‌نویسی کامپیوتر و زبان برنامه‌نویسی C++ است.

عموم مردم در مورد اعمالی که کامپیوترها انجام می‌دهند آشنایی دارند. با استفاده از این کتاب با دستوراتی آشنا خواهید شد که اعمال ویژه‌ای در یک موضوع خاص انجام می‌دهند و کامپیوتر انجام این دستورات را برعهده دارد. نرم‌افزار (دستوراتی که بصورت کد نوشته شده و کامپیوتر آن دستورات را انجام داده و در مورد آنها تصمیم‌گیری می‌کند) کامپیوتر را کنترل می‌کند (غالبا از آن بعنوان سخت‌افزار یاد می‌شود). C++ یکی از ابزارهای قدرتمند در توسعه نرم‌افزار است. در این کتاب به معرفی اصول برنامه‌نویسی در زبان C++ استاندارد شده در ایالات متحده بر اساس ANSI[1] و استاندارد جهانی ISO[2] می پردازیم.

استفاده از کامپیوترها مستلزم سعی و کوشش زیادی است. در عصری که هزینه‌ها مرتباً در حال افزایش است کاهش هزینه سخت‌افزار و نرم‌افزار که بسرعت در حال توسعه است می‌تواند جالب توجه باشد. کامپیوترهایی که 25 الی 30 سال پیش فضای زیادی از اطاق‌ها را در بر می‌گرفتند و هزینه‌های آنها بالغ بر میلیون‌ها دلار می‌شد، امروزه بر روی یک سطح تراشه سیلیکونی که به اندازه یک ناخن دست است و فقط چند دلار قیمت دارد جا گرفته‌اند. سیلیکون یکی از فراوان‌ترین فلزات موجود بر روی زمین است و عموماً جزء اجزای ترکیبی ماسه یا شن می‌باشد. تکنولوژی تراشه سیلیکونی باعث ساخته شدن کامپیوترهای مقرون به صرفه شده و بیش از ۲۰۰ میلیون کامپیوتر موجود در سرتاسر جهان از این تکنولوژی استفاده می‌کنند، که مورد استفاده عمومی در زمینه‌های تجاری، صنعتی، دولتی و امور روزمره می‌باشند.

در طول سالیان گذشته، بسیاری از برنامه‌نویسان شروع به یادگیری روشی بنام برنامه‌نویسی ساخت‌یافته کردند. دراین کتاب به آموزش برنامه‌نویسی ساخت‌یافته و همچنین مبحث جالب برنامه‌نویسی *شی‌گرا* (*Object-Oriented*) می‌پردازیم. چرا به آموزش هر دو روش می‌پردازیم؟ برنامه‌نویسی شی‌گرا کلید برنامه‌نویسی دهه بعد است. در این کتاب شی‌های متعددی ایجاد و با آنها کار خواهید کرد. در ضمن مطالعه ساختار داخلی شی‌ها متوجه خواهید شد که در ایجاد این شی‌ها از تکنیک‌های برنامه‌نویسی ساخت‌یافته استفاده شده است. همچنین دستکاری منطقی شی‌ها در بهترین حالت با برنامه‌نویسی ساخت‌یافته صورت می‌گیرد.

[1] American Natiomnal Standards Institute
[2] International Organization for Standardization

۱-۲ کامپیوتر چیست؟

کامپیوتر وسیله‌ای است که توانایی انجام محاسبات و تصمیم‌گیری‌های منطقی با سرعت میلیون‌ها و حتی بیلیون‌ها برابر سریع‌تر از یک انسان را دارد. برای مثال امروزه بیشتر کامپیوترهای شخصی می‌توانند صدها میلیون دستور در هر ثانیه انجام دهند. یک ماشین حساب بایستی تمام عمر کار کند تا بتواند همان عدد محاسبه شده توسط یک کامپیوتر شخصی را که در عرض یک ثانیه انجام داده به دست بیاورد. امروزه سوپر کامپیوترها توانایی انجام صدها بیلیون دستور در هر ثانیه را دارند در حالیکه صدها هزار نفر با استفاده از ماشین حساب این عمل را در طول یک سال می‌توانند انجام دهند. کامپیوترهایی که توانایی انجام تریلیون‌ها دستور در هر ثانیه دارند در آزمایشگاه‌های تحقیقاتی مورد استفاده می‌باشند. کامپیوترها پردازش داده‌ها را تحت کنترل تعدادی از دستورات که برنامه‌های کامپیوتری نامیده می‌شوند، انجام می‌دهند و این برنامه‌ها توسط اشخاصی بنام *برنامه‌نویس* نوشته می‌شوند.

انواع متفاوتی از قطعات همانند صفحه کلید، صفحه نمایش، دیسک‌ها، حافظه و واحدهای پردازش، که یک سیستم کامپیوتری را تشکیل می‌دهند بنام *سخت‌افزار* شناخته می‌شوند. برنامه‌های کامپیوترئی که بتوانند بر روی یک کامپیوتر اجرا شوند بنام *نرم‌افزار* شناخته می‌شوند. هزینه‌های سخت‌افزاری در سال‌های اخیر کاهش یافته و کامپیوترهای شخصی بعنوان یک ابزار مناسب در دسترس قرار گرفته‌اند، ولی متأسفانه هزینه‌های توسعه نرم‌افزار افزایش یافته و برنامه‌نویسان برنامه‌های کاربردی قویتر و پیچیده‌تری ارائه می‌کنند بدون اینکه قادر باشند تکنولوژی توسعه نرم‌افزار را بهبود بخشند. در این کتاب سعی شده است تا بتوانید روش‌های توسعه نرم‌افزار را با استفاده از کاهش هزینه‌های نرم‌افزار از طریق به کارگیری روش‌های برنامه‌نویسی ساخت‌یافته از بالا به پایین (top-down)، برنامه‌های مبتنی بر شی، برنامه‌نویسی شی‌گرا، طراحی شی‌گرا فرا بگیرید.

۱-۳ سازماندهی کامپیوتر

صرفنظر از تفاوت‌های فیزیکی، هر کامپیوتر به شش قسمت منطقی تقسیم می‌شود که عبارتند از:

۱- واحد ورودی (Input unit):

این قسمت "واحد دریافت" برای بدست آوردن اطلاعات (داده و برنامه کامپیوتری) از طریق انواع وسایل ورودی و قرار دادن این اطلاعات در دسترس سایر واحدها برای پردازش اطلاعات می‌باشد. بیشتر اطلاعات ورودی از طریق صفحه کلید و ماوس دریافت می‌شوند. در آینده ممکن است حجم زیادی از اطلاعات بفرم ویدئوی دریافت شوند.

۲- واحد خروجی (Output unit):

این قسمت وظیفه "حمل" را برعهده دارد. این واحد اطلاعات پردازش شده توسط کامپیوتر را دریافت کرده و به انواع وسایل خروجی منتقل می‌کند تا در خارج از کامپیوتر مورد استفاده قرار گیرد. بیشتر اطلاعات خروجی بر روی صفحات نمایش و چاپ بر روی کاغذ منتقل می‌شوند یا از اطلاعات خروجی برای کنترل سایر دستگاه‌ها استفاده می‌شود.

۳- واحد حافظه (Memory unit):

این قسمت "انبار" کامپیوتر محسوب می‌شود و دارای سرعت دسترسی و ظرفیت نسبتاً بالایی است. اطلاعات وارد شده از واحد ورودی در این قسمت نگهداری می‌شوند. همچنین واحد حافظه می‌تواند اطلاعات پردازش شده را در خود نگهداری کند، تا زمانیکه اطلاعات بتوانند بر روی دستگاه خروجی (به وسیله واحد خروجی) قرار گیرند. واحد حافظه اغلب به عنوان حافظه یا حافظه اولیه نامیده می‌شود.

٤- واحد محاسبه و منطق (ALU):

این قسمت "واحد ساخت" کامپیوتر است. این بخش مسئولیت انجام اعمال محاسباتی همانند جمع، تفریق، ضرب و تقسیم را برعهده دارد. همچنین شامل مکانیزم‌های تصمیم‌گیری می‌باشد. بطور مثال به کامپیوتر این امکان را می‌دهد که محتویات دو محل متفاوت از حافظه را باهم مقایسه کرده و تعیین کند که آیا برابرند یا خیر.

٥- واحد پردازش مرکزی (CPU):

این قسمت واحد "اجرائی" کامپیوتر است. این قسمت وظیفه هماهنگ کردن کامپیوتر و مسئولیت نظارت بر نحوه انجام عملیات توسط سایر قسمت‌ها را برعهده دارد. CPU به واحد ورودی اعلان می‌کند که در چه زمانی می‌بایستی اطلاعات به واحد حافظه وارد شده و به ALU اعلان می‌کند در چه زمانی اطلاعات از حافظه برداشته و بکار گرفته شوند و به واحد خروجی اعلان می‌کند که در چه زمانی اطلاعات از حافظه به واحد خروجی مشخص شده ارسال شوند.

٦- واحد ذخیره‌سازی ثانویه:

این قسمت "انبار" کامپیوتر است که دارای طول عمر زیاد و ظرفیت بالا می‌باشد. برنامه‌ها یا داده‌ها تا زمانیکه بر روی وسایل ذخیره‌سازی ثانویه (همانند دیسک‌ها) ذخیره نشوند، نمی‌توانند بدرستی بکار گرفته شوند. اطلاعات قرار گرفته بر روی واحد ذخیره‌سازی ثانویه بدفعات زیاد نسبت به حافظه می‌توانند مورد دستیابی قرار گیرند. هزینه وسایل ذخیره‌سازی ثانویه نسبت به حافظه اولیه بسیار کمتر است.

٤-١ تکامل سیستم عامل

زمانی کامپیوترها فقط می‌توانستند یک عمل یا یک وظیفه را در هر زمان انجام دهند اینحالت در کامپیوترها به عنوان پردازش *دسته‌ای تک‌کاربره* (*single user batch processing*) معروف است. کامپیوتر در هر زمان توانایی اجرای یک برنامه در زمان پردازش را دارد. در این سیستم‌ها کاربران معمولاً کارهایی که

می‌خواستند انجام دهند بر روی کارت پانچ قرار می‌دادند و به کامپیوتر مرکزی ارائه می‌کردند. کاربران اغلب ساعت‌ها و حتی روزها منتظر جواب می‌شد. نرم‌افزارهای سیستم که معروف به سیستم عامل هستند به منظور استفاده آسانتر از کامپیوترها توسعه پیدا کردند. سیستم‌های عامل قدیمی، مدیریت انتقالی بین وظایف محوله را انجام می‌دادند. هنگامی که کامپیوترها قویتر و کاراتر شدند، سیستم‌های تک کاربره در استفاده از منابع سیستم دیگر کارایی قابل قبولی نداشتند. برای مثال، بایستی تعداد متنوعی از وظایف با استفاده از اشتراک منابع برای استفاده بهینه از کامپیوتر مورد استفاده قرار گیرد که بنام *Multiprogramming* نامیده می‌شود. Multiprogramming چندین عملیات را بصورت همزمان در یک کامپیوتر انجام می‌دهد (این قابلیت با عنوان *Multiprocessing* نیز شناخته می‌شود). کامپیوتر با استفاده از اشتراک منابع در میان انواع وظایف به فعالیت خود ادامه می‌دهد. اما هنوز هم در این سیستم‌های عامل، بایستی کاربران ساعت‌ها در انتظار باقی می‌ماندند.

در دهه ۱۹۶۰ چندین گروه از صنایع و دانشگاه‌ها پیش گام توسعه سیستم‌های عامل *اشتراک زمانی* (*Timesharing*) شدند. اشتراک زمانی یک حالت خاص از Multiprogramming می‌باشد، که در آن کاربران از طریق یک ترمینال که نوعاً یک صفحه کلید و صفحه نمایش می‌باشد به کامپیوتر دسترسی داشتند. در نمونه واقعی کامپیوتری که از سیستم اشتراک زمانی استفاده می‌کند ممکن است یک دوجین یا حتی صدها کاربر بصورت مشترک از کامپیوتر استفاده کنند. کامپیوتر نمی‌تواند به درخواست‌های همزمان کاربران واکنش نشان دهد. در اینحالت کامپیوتر یک قسمت از کار یک کاربر را انجام داده و سپس سرویس را به کاربر بعدی انتقال می‌دهد. کامپیوتر این عمل را بسیار سریع انجام می‌دهد و ممکن است به چندین کاربر در هر ثانیه سرویس ارائه کند. در اینحالت کاربران گمان می‌کنند که برنامه‌ها بصورت همزمان اجرا می‌شوند. مزیت اشتراک زمانی این است که به درخواست کاربر سریعاً واکنش نشان داده می‌شود.

۱-۵ محاسبات شخصی، محاسبات توزیع شده و سرویس دهنده‌ها/ سرویس گیرنده‌ها

در سال ۱۹۷۷، کامپیوترهای اپل (Apple)، نماد محاسبات شخصی بودند. کامپیوترها به تدریج ارزان شدند تا مردم آنها را خریداری کرده و در کارهای شخصی یا تجاری مورد استفاده قرار دهند. در سال ۱۹۸۱ شرکت IBM که بزرگترین فروشنده کامپیوتر در جهان است، کامپیوترهای شخصی IBM را به بازار معرفی کرد. بسرعت محاسبات شخصی در تجارت، صنایع و مراکز دولتی وارد شد. اما این کامپیوترها هنوز هم بفرم واحدهای منفرد عمل می‌کردند. کاربران کارهای خود را بر روی سیستم خود انجام می‌دادند و سپس نتایج را بر روی دیسک منتقل می‌کردند و آنرا به اشتراک می‌گذاشتند. با اتصال چندین سیستم به یکدیگر شبکه تشکیل داده شد. شبکه‌های محلی (LAN) از این نوع سازماندهی می‌باشند. این فرآیند سبب هدایت بسوی محاسبات توزیع شده در سازماندهی محاسباتی گردید. کامپیوترهای شخصی بقدر کافی قدرت پیدا کرده

بودند که می‌توانستند محاسبات جداگانه چندین کاربر را انجام داده و وظایف ارتباطی و عبور اطلاعات بصورت الکترونیکی را فراهم نمایند.

امروزه کامپیوترهای شخصی نسبت به کامپیوترهای دهه قبل چندین میلیون برابر، قدرت بیشتر پیدا کرده‌اند. ماشین‌های قدرتمند رومیزی که *ایستگاه کاری* (*Workstation*) نامیده می‌شوند، توانایی بسیار زیادی در ارائه سرویس به کاربران با نیازهای متفاوت دارند.

اطلاعاتی که حالت اشتراکی دارند در کامپیوترهای شبکه موسوم به *سرویس‌دهنده* (*Server*) قرار می‌گیرند. این کامپیوترها اطلاعات و برنامه‌ها را در خود نگهداری می‌کنند که ممکن است توسط *س‌رویس‌گیرنده‌ها* (*Clients*) که در سرتاسر جهان توزیع شده‌اند مورد استفاده قرار گیرند، از اینرو عبارت *سرویس‌دهنده/سرویس‌گیرنده* (*Server/Client*) وارد صحنه گردید. زبان‌های C و C++ به عنوان زبان‌های برنامه‌نویسی، برای نوشتن نرم‌افزار سیستم عامل برای کامپیوترهای شبکه و کاربردهای توزیع شده Server/Client انتخاب شده‌اند، امروزه سیستم‌های عامل پرطرفدار همانند UNIX، Linux، Solaris، MacOs، Windows 2000 و Windows XP دارای قابلیت‌های فراوانی هستند که در مورد آنها صحبت خواهیم کرد.

۶-۱ تاریخچه اینترنت و www

در اواخر دهه ۱۹۶۰، پورفسور H.M.Deitel از دانشجویان فارغ‌التحصیل دانشگاه MIT بود. پروفسور Deitel بر روی پروژه Mac[1] دانشگاه MIT که سبب پیدایش ARPA[2] شده کارکرده است. ARPA میزبان کنفرانسی شد که میهمانان آن مجموعه‌ای از پدید آورندگان ARPA در دانشگاه Illinois بودند و در آن به بحث و بررسی مباحث مختلف پرداخته شد. در این کنفرانس طرح شبکه کردن، کامپیوترهای اصلی دانشگاه‌های سهیم در پروژه ARPA مطرح گردید. کامپیوترهای متصل شده با خطوط ارتباطی با سرعت کمتر 56 kbps کار می‌کردند (kbps 1 معادل، 1024 بیت در هر ثانیه است)، در آن زمان بیشتر مردم (کسانی که به شبکه دسترسی داشتند) از طریق خطوط تلفن با سرعت 110 بیت در هر ثانیه به کامپیوترها متصل می‌شدند. در این کنفرانس در مورد مباحث گوناگونی صحبت شد و سرانجام ARPA به ARPAnet تغییر نام داد که پدر بزرگ اینترنت است.

گروه‌های مختلف بر روی طرح اولیه به روش‌های گوناگونی کار کردند. اگر چه ARPAnet امکان تحقیق به محققان خود را بر روی کامپیوترهای شبکه می‌داد، اما اصلی‌ترین مزیت آن بهبود قابلیت برای ارتباط آسان و سریع بود که امروزه بنام پست الکترونیکی (e.mial) شناخته می‌شود. این قابلیت امروزه نیز در

۱- هم اکنون آزمایشگاه علوم کامپیوتر و منزلگاه کنسرسیوم World Wide Web است.

۲- Advanced Research Project Agency of Department

اینترنت در زمینه پست الکترونیکی و انتقال فایل در میان میلیون‌ها نفر در سرتاسر جهان بکار گرفته می‌شود. شبکه طراحی شدهٔ در آن زمان فاقد یک کنترل مرکزی بود. به این دلیل که اگر بخشی از شبکه از مدار خارج می‌شد، مابقی بخش‌های شبکه هنوز هم قادر به ارسال و دریافت بسته‌های اطلاعاتی از طریق مسیرهای جایگزین بودند.

پروتکل (مجموعه قوانین) برقراری ارتباط بر روی شبکه ARPAnet امروزه بنام *TCP* (*Transmission Control Protocol*) شناخته می‌شود. این پروتکل سبب می‌شود که پیغام‌ها با دقت و به درستی از سوی فرستنده به گیرنده ارسال شوند. برای تشخیص بخش‌های مقابل، شبکه ARPAnet موجب توسعه پروتکل اینترنت یا IP (*Internet Protocol*) شد که بدنبال آن واقعیت "شبکه‌ای از شبکه‌ها" تحقق پیدا کرد و معماری جاری در اینترنت شد. مجموعه‌ای از این پروتکل‌ها بعنوان TCP/IP شناخته می‌شود.

وب گسترده جهانی (*World Wide Web*) به کاربران کامپیوتر امکان می‌دهد تا مستندات مبتنی بر مولتی مدیا را یافته و به آنها نگاه کنند (مستنداتی متشکل از متن، گرافیک، انیمیشن، صوت یا ویدئو). در سال ۱۹۸۹، پورفسور Tim Berners-Lee از گروه CERN (سازمان اروپا در زمینه تحقیقات هسته‌ای) شروع به توسعه تکنولوژی، در زمینه به اشتراک گذاری اطلاعات از طریق فوق لینک‌ها در مستندات متنی کرد. اینکار بر مبنای زبان جدیدی بنام SGML صورت گرفت (استانداردی برای تبادل اطلاعات)، که Berners-Lee آنرا HTML[1] (*HyperText Markup Language*) نامید. البته Lee پروتکل‌های ارتباطی برای سیستم اطلاعاتی فوق متن جدید خود نوشت که او از آن بعنوان World Wide Web نام برد. امروزه اینترنت و WWW از بخش‌های اصلی و ضروری در زندگی انسانها شده‌اند. در گذشته، بیشتر برنامه‌های کامپیوتری روی یک سیستم منفرد به اجرا در می‌آمدند (کامپیوترهای که به کامپیوتر دیگری متصل نبودند). امروزه برنامه‌های کامپیوتری می‌توانند برای برقراری ارتباط مابین میلیون‌ها کامپیوتر نوشته شوند.

۷-۱ زبان ماشین، زبان اسمبلی و زبان سطح بالا

برنامه‌نویس دستورات خود را می‌تواند در انواع متفاوتی از زبان‌های برنامه‌نویسی بنویسد. تعدادی از این زبان‌ها به صورت مستقیم توسط کامپیوتر درک می‌شوند و تعداد دیگری نیاز به ترجمه دارند تا قابل فهم برای کامپیوتر شوند. امروزه صدها زبان کامپیوتری مورد استفاده می‌باشند، که می‌توان آنها را به سه دسته تقسیم کرد:

۱- زبان ماشین (Machine Languages)

۲- زبان اسمبلی (Assembly Languages)

۱- زبان نشانه‌گذاري فوق متن

۳- زبان‌های سطح بالا (High-Level Languages)

هر کامپیوتری می‌تواند بطور مستقیم فقط زبان ماشین خود را درک کند. زبان ماشین، زبان ذاتی و منحصر بفرد یک کامپیوتر می‌باشد و به هنگام طراحی سخت‌افزار کامپیوتر تعریف می‌شود. زبان ماشین عموماً شامل رشته‌ای از اعداد است و موجب می‌شود که کامپیوتر عملیات اصلی را که در ارتباط با خود است درهر بار راه‌اندازی اجرا نماید. زبان ماشین، وابسته به ماشین می‌باشد (زبان ماشین یک دستگاه فقط بر روی همان نوع از ماشین اجرا می‌شود). درک زبان ماشین برای انسان طاقت فرسا و بسیار مشکل است. برای مثال می‌توانید به زبان ماشین که در قسمت زیر آورده شده توجه کنید، این برنامه اضافه کار را بر مبنای حقوق محاسبه و نتیجه بدست آمده را در grosspay ذخیره می‌کند.

```
+1300042774
+1400593419
+1200274027
```

زمانیکه کامپیوترها مورد استفاده عموم قرار گرفتند، مشخص شد برنامه‌نویسی زبان ماشین برای بسیاری از برنامه‌نویسان خسته کننده و ملالت‌آور است. در عوض، بکار بردن رشته‌ای از اعداد که کامپیوتر بتواند بصورت مستقیم آنرا درک کند، برنامه‌نویسان از عبارات کوتاه شده زبان انگلیسی برای فهماندن عملیات ابتدایی به کامپیوتر استفاده کردند. این عبارات مخفف شده شبیه زبان انگلیسی، مبنای زبان اسمبلی هستند. برنامه‌های مترجم بنام اسمبلر مشهور می‌باشند که زبان اسمبلی را بزبان ماشین ترجمه می‌کنند. قطعه برنامه‌ای که در قسمت پایین آورده شده همان عملیات بالا را انجام می‌دهد منتهی با استفاده از زبان اسمبلی که نسبت به زبان ماشین از وضوح (قابل فهم) بیشتری برخوردار است.

```
LOAD BASEPAY
ADD OVERPAY
STORE GROSSPAY
```

اگر چه این کد برای انسان از وضوح بیشتری برخوردار است اما برای کامپیوتر تا زمانی که به زبان ماشین ترجمه نشود معنی ندارد. زبان اسمبلی باعث افزایش سرعت برنامه‌نویسی شد اما هنوز هم مستلزم دستورات فراوانی برای انجام یک عمل ساده بود. برای افزایش سرعت برنامه‌نویسی زبان‌های سطح بالا توسعه پیدا کردند. که با استفاده از یک عبارت می‌توانند وظایف و اعمال وسیع‌تری را انجام دهند. برنامه‌های مترجم که وظیفه تبدیل زبان‌های سطح بالا به زبان ماشین را برعهده دارند کامپایلر نامیده می‌شوند. زبان‌های سطح بالا این امکان را به برنامه‌نویس می‌دهند که دستورات مورد نیاز خود را تقریباً مانند زبان انگلیسی و عملیات ریاضی را به صورت روزمره بنویسد.

```
grossPay = basePay + overTimePay
```

واضح است که زبان‌های سطح بالا نسبت به زبان‌های ماشین یا اسمبلی از محبوبیت بیشتری در نزد برنامه‌نویسان برخوردارند. ویژوال بیسیک به صورت وسیع مورد استفاده می‌باشد و از جمله زبان‌های سطح بالا به شمار می‌آید. عمل کامپایل کردن زبان سطح بالا به زبان ماشین می‌تواند وقت زیادی از کامپیوتر را بگیرد. برنامه‌های مفسر (Interpreter) توسعه یافته می‌توانند به صورت مستقیم برنامه‌های زبان‌های سطح بالا را بدون نیاز به کامپایل به زبان ماشین تبدیل کنند. اگر چه برنامه‌های مفسر نسبت به برنامه‌های کامپایلر آهسته‌تر عمل می‌کنند، اما برنامه‌های مفسر فوراً شروع به فعالیت می‌کنند بدون اینکه تأخیرهای ذاتی از عمل کامپایل را در خود داشته باشند.

۸-۱ تاریخچه C و C++

زبان C++ توسعه یافته زبان C است که از دو زبان برنامه‌نویسی قبلی، بنام‌های BCPL و B منشعب شده است. زبان BCPL در سال 1967 توسط Martin Richards بعنوان زبانی برای نوشتن نرم‌افزار سیستم‌های عامل و کامپایلرها طراحی شده بود.آقای Ken Thompson بسیاری از ویژگیهای زبان B خود را از BCPL اقتباس کرد و از B برای ایجاد نسخه‌های اولیه سیستم عامل UNIX در آزمایشگاهای Bell در سال 1970 بر کامپیوتر DEC PDP-7 استفاده شد. هر دو زبان BCPL و B از نوع زبانهای بدون نوع (typeless) هستند، به این معنی که هر ایتم داده یک "کلمه" در حافظه اشغال می‌کند و مسئولیت رسیدگی به دادها به عهده برنامه نویس خواهد بود.

زبان C از زبان B و توسط Dennis Ritchie در آزمایشگاهای شرکت Bell توسعه یافت و برای اولین بار بر روی کامپیوتر DEC PDP-11 در سال 1972 پیاده سازی گردید. زبان C از مفاهیم اساسی BCPL و B سود می‌برد در حالیکه دارای قابلیت تعریف *نوع داده* (*data type*) و ویژگیهای دیگر بود. زبان C در بدو شروع بکار بطور گسترده‌ای بعنوان زبان توسعه‌دهنده سیستم عامل UNIX بکار گرفته شد. امروزه، اکثر سیستم‌های عامل توسط زبان‌های C یا C++ یا ترکیبی از هر دو نوشته شده‌اند. هم اکنون C بر روی بیشتر کامپیوتر پیدا می‌شود. زبان C، زبان مستقل از سخت‌افزار است. اگر در زمان طراحی دقت کافی بخرج داده شود، می‌توان برنامه‌های C را که از قابلیت حمل (portable) برخودار هستند بر روی اکثر کامپیوترها به اجرا در آورد.

در اواخر دهه 1970، زبان C توسعه پیدا کرد و بنام‌های "C تجاری"،"C کلاسیک"، و " Keringhan and Ritchie C" معروف شد. کتاب "زبان برنامه‌نویسی C" که توسط انتشارات Prentice-Hall در سال 1978 منتشر شد تاثیر بسیار زیادی در گسترش این زبان بازی کرد.

بکارگیری زبان C بر روی مجموعه وسیعی از انواع کامپیوترها (گاهی اوقات از این مطلب بعنوان platform یاد می‌شود) موجب شده تا نسخه‌های متعدی از آن بوجود آید (متاسفانه). با اینکه این نسخه‌ها شبیه هم بودند، اما گاهی اوقات عدم سازگاری مابین آنها رخ می‌داد. این عدم سازگاری یکی از جدی‌ترین مشکلات برنامه‌نویسانی بود که می‌خواستند برنامه‌های قابل حملی بنویسند که بر روی چندین پلات‌فرم به اجرا درآید. در چنین وضعیتی وجود یک نسخه استاندارد C احساس گردید. در سال 1983، کمیته استاندارد X3J11 که تحت نظارت کمیته ملی استاندارد کامپیوتر و پردازش اطلاعات آمریکا[1](ANSCIP) بوجود آمده بود، یک تعریف غیر مبهم از زبان مستقل از ماشین ارائه کرد. در سال 1989، استاندارد رشد پیدا کرده بود و ANSI با همراهی ISO[2] زبان C را در سرتاسر جهان استاندارد کردند. مستند استاندارد در سال 1990 منتشر شد و از آن بعنوان ANSI/ISO 9899:1990 یاد می‌شود. ویرایش دوم کتاب "زبان برنامه‌نویسی C" در سال 1988، چاپ و به نام ANSI C، نامیده شد که هم اکنون در سرتاسر جهان بکار گرفته می‌شود.

قابلیت حمل

بدلیل اینکه زبان C++ استاندارد شده است، مستقل از سخت افزار است و بطرز گسترده ای در دسترس می باشد، غالبا برنامه‌های نوشته شده به زبان C++ را می‌توان با کمی اصلاح و حتی بدون هیچ گونه تغییری، بر روی انواع مختلفی از سیستم‌های کامپیوتری به اجرا در آورد.

زبان C++ بسط یافته زبان C است،که توسط Bjarne Stroustryp در اوایل 1980 و در آزمایشگاه‌های Bell ابداع گردید. زبان C++ حاوی برخی از ویژگیهای C است، اما مهمترین ویژگی و قابلیت این زبان در برنامه‌نویسی شی گرا بودن آن است.

این ویژگی، انقلابی در جامعه نرم‌افزاری بوجود آورد. در چنین حالتی تولید نرم‌افزار بسرعت، با دقت و اقتصادی‌تر صورت می‌گیرد. شی‌ها *کامپونت‌های* نرم‌افزاری با قابلیت *استفاده مجدد* هستند که ایتم‌های حقیقی در دنیا را مدل سازی می‌کنند. توسعه دهنده‌گان نرم‌افزار به این نتیجه رسیده‌اند که بهره گیری از روش مدولار و طراحی شی‌گرا و پیاده سازی به این روش‌ها می‌تواند در بهره‌وری گروه‌های توسعه‌دهنده نرم‌افزاری در مقایسه با تکنیک‌های متداول قدیمی‌تر، همانند برنامه‌نویسی ساخت‌یافته بسیار موثر باشد. درک برنامه‌های شی‌گرا، اصلاح و تغییر آنها راحتر است.

زبانهای شی‌گرای متعددی تا بدین روز پدید آمده‌اند، زبان‌های همانند Smalltalk، که توسط PARC[3] ابداع شده است. زبان Smalltalk یک زبان شی‌گرای محض است، که هر چیزی در آن یک شی می‌باشد. از

[1] American NationalStandards Committee on Computers and Information Processing (X3)
[2] International Standards Organization
[3] Xerox's Palo Alto Research Center

سوی دیگر C++ از جمله زبان‌های هیبرید می‌باشد، به این معنی که می‌توان در این زبان برنامه‌ای نوشت که شبیه C یا شی‌گرا باشد، یا اینکه ترکیبی از هر دو حالت را در بر گیرد.

۹-۱ کتابخانه استاندارد C++

برنامه‌های C++ متشکل از قسمت‌های بنام کلاس‌ها و توابع هستند. می‌توانید برحسب نیاز هر قسمت را به فرم یک برنامه C++ برنامه‌نویسی کنید. با این همه, اکثر برنامه‌نویسان C++ از مزیت کلکسیون‌های غنی از کلاس‌ها و توابع در کتابخانه استاندارد C++ بهره می‌برند. از اینرو, واقعاً دو بخش آموزشی در جهان C++ وجود دارد. بخش اول یادگیری خود زبان C++ است و بخش دوم نحوه استفاده از کلاس‌ها و توابع موجود در کتابخانه استاندارد C++. در سرتاسر این کتاب در ارتباط با تعدادی از این کلاس‌ها و توابع صحبت خواهیم کرد. خواندن کتاب The Standard C Library, نوشته PJ.Plauger را برای کسانی که می‌خواهند درک عمیقی از توابع کتابخانه ANSI C که در برگیرنده C++ نیز می‌باشد و در آن مطالبی در زمینه نحوه پیاده‌سازی و نحوه استفاده از این توابع در ایجاد کدهای قابل حمل وجود دارد, توصیه می‌کنیم. معمولاً کتابخانه‌های استاندارد توسط سازندگان کامپایلر تدارک دیده می‌شوند. البته کتابخانه‌های کلاس با مقاصد خاص نیز وجود دارند که توسط سازندگان مستقل نرم‌افزار تهیه می‌شوند.

مهندسی نرم‌افزار

از روش «ساخت- بلوکی» در ایجاد برنامه‌ها استفاده کنید. از اختراع مجدد چرخ اجتناب کنید. تا حد امکان از قسمت‌های موجود استفاده کنید. به این روش, استفاده مجدد از نرم‌افزار می‌گویند, که یکی از اهداف برنامه‌نویسی شی‌گرا می‌باشد.

مهندسی نرم‌افزار

به هنگام برنامه‌نویسی در C++, از بلوک‌های زیر استفاده خواهید کرد: کلاس‌ها و توابع موجود در کتابخانه استاندارد C++, کلاس‌ها و توابعی که خودتان ایجاد می‌کنید، کلاس‌ها و توابعی که در کتابخانه‌های دیگر وجود دارند.

در کل کتاب با نکاتی در ارتباط با مهندسی نرم‌افزار مشاهده خواهید کرد که به توصیف مفاهیم موثر در بهبود معماری و کیفیت سیستم نرم‌افزاری می‌پردازند. همچنین به برجسته کردن نکات دیگری شامل برنامه‌نویسی ایده‌آل (برای کمک به شما در نوشتن برنامه‌هایی که واضح بوده, درک آنها آسانتر باشد, نگهداری و استفاده راحت‌تری داشته باشند و بتوان خطاهای آنها را به راحتی یافته و اصلاح کرد), خطاهای برنامه‌نویسی (مشکلاتی که باید برای اجتناب از آنها هوشیار باشید), کارایی (تکنیک‌های برنامه‌نویسی که سبب اجرای سریعتر و مصرف کمتر حافظه می‌شوند), قابلیت حمل (تکنیک‌هایی که به کمک آنها می‌توان برنامه‌هایی نوشت که با کمی تغییر یا هیچ تغییری, بر روی انواع کامپیوترها اجرا شوند) و اجتناب از خطا

(تکنیک‌های حذف خطا از برنامه‌ها و روش نوشتن برنامه‌های بدون خطا از همان ابتدای کار) است. تمام این نکات فقط نقش راهنما و هدایت‌کننده دارند.

یکی از مزایا ایجاد توابع و کلاس‌های متلعق بخود این است که دقیقاً از نحوه عملکرد آنها مطلع هستیم و می‌توانیم به بررسی کد C++ آنها بپردازیم. عیب این روش در زمان‌بر بودن و پیچیدگی است که در طراحی, توسعه و نگهداری توابع و تلاش‌های جدید بوجود می‌آید.

کارائی

با استفاده از توابع و کلاس‌های کتابخانه استاندارد C++, بجای نوشتن نسخه‌هایی از آنها, می‌توانید کارایی برنامه را افزایش دهید چرا که آنها بدقت و با توجه به کارایی نوشته شده‌اند. همچنین این تکنیک زمان توسعه و ایجاد برنامه را کوتاهتر می‌سازد.

قابلیت حمل

قابلیت حمل با استفاده از توابع و کلاس‌های کتابخانه استاندارد C++ بجای نوشتن توابع و کلاس‌های متعلق به خود, قابلیت حمل برنامه افزایش و بهبود می‌یابد چرا که این توابع و کلاس‌های در هر پیاده‌سازی C++ وجود دارند.

۱۰-۱ تاریخچه جاوا

بسیاری از افراد بر این باورند که آینده در اختیار ریزپردازنده‌های هوشمند خواهد بود. با در نظر گرفتن این مطلب، شرکت Sun Microsystems یک تیم تحقیقاتی با نام کد Green در سال 1990 تاسیس کرد. نتیجه پروژه که مبتنی بر زبان‌های C و C++ بود توسط James Gosling بنام Oak نامیده شد. پس ملاقات افراد تیم Sun در یک کافه محلی بر سر نام *جاوا* (*Java*) به توافق رسیدند.

اما پروژه Green با مشکلاتی مواجه شد. بازار قطعات هوشمند مطابق با آنچه که شرکت Sun انتظار داشت رشد نکرد. بدتر از آن قراردادی که شرکت Sun بر سر آن رقابت می‌کرد به یک شرکت دیگر واگذار گردید. از اینرو پروژه در وضعیت خطرناک لغو قرار گرفت. از بخت بلند، در سال 1993، گشت و گذار در *وب گسترده جهانی(WWW)* از محبوبیت بسیار زیادی در بین مردم برخوردار شده بود. بنابر این اهالی Sun بلافاصله متوجه کاربرد جاوا و پتانسیل‌های آن در ایجاد *محتویات دینامیک* بر صفحات وب شدند.

شرکت Sun در ماه می 1995 به عرضه تجاری جاوا پرداخت. بلافاصله، جاوا نظر بسیاری از مراکز تجاری را بخود جلب کرد، چراکه علاقه عجیبی به وب گسترده جهانی در نزد مردم پیدا شده بود. هم اکنون از جاوا برای ایجاد صفحات وب با قابلیت دینامیکی و تعاملی، توسعه برنامه‌های کاربردی در مقیاس گسترده، به منظور افزایش کارائی و عملکرد سرویس‌دهنده‌های وب (کامپیوترهائی که محتویات بنمایش درآمده در مرورگرهای وب را فراهم می‌آورند) و همچنین برنامه‌نویسی دستگاه‌های همانند تلفن‌های موبایل، فراخوان و

PDA[1] استفاده می‌شود. در سال 1995، شاهد توسعه جاوا توسط شرکت Sun Microsystems بودیم. در نوامبر 1995، کنفرانسی در بوستون در ارتباط با اینترنت برگزار شد. فردی از طرف شرکت Sun Microsystems مقاله‌ای در ارتباط با جاوا ارائه کرد. در حین بیان مقاله، برای ما مشخص شد که جاوا در حال کار بر روی توسعه صفحات وب با قابلیت‌های مولتی مدیا و تعاملی است. چندی نگذشت که متوجه قابلیت‌های بسیار زیاد این زبان شدیم. دانستیم که جاوا زبان مناسبی برای دانشجویان سال اول در مبحث آموزش زبان برنامه‌نویسی است، چراکه این زبان شامل مباحث گرافیگ، تصویر، انیمیشن، صدا، ویدئو، پایگاه داده، شبکه، چندنخی (Multithreading) و محاسبات همروند است.

علاوه بر اینها، جاوا بطور بارزی تبدیل به زبان انتخابی به منظور پیاده سازی نرم‌افزار دستگاه‌های ارتباطی بر روی یک شبکه شده است (همانند تلفن‌های موبایل، فراخوان، و PDA). از اینکه زمانی فرا برسد که ضبط صوت ودستگاه‌های دیگر منزلتان با استفاده از تکنولوژی جاوا تشکیل یک شبکه را بدهند تعجب نکنید.

۱-۱۱ فرترن، کوبول و پاسکال

تا کنون صدها زبان سطح بالا ایجاد شده اما فقط تعدادی از آنها موفقیت قابل قبولی بدست آورده‌اند.

FORTRAN (*FORmula TRANslator*) توسط شرکت IBM در بین سال‌های ۱۹۵۴ و ۱۹۵۷ ایجاد شده و در کاربردهای عملی و مهندسی که نیاز به محاسبات پیچیده ریاضی دارد بکار گرفته می‌شود. فرترن هنوز هم به صورت گسترده‌ای مورد استفاده قرار می‌گیرد. مخصوصاً در کاربردهای مهندسی.

COBOL (*Common Business Oriented Language*) توسط گروهی از سازنده‌های کامپیوتر، دولت و کارخانه‌هایی که از کامپیوتر استفاده می‌کردند در سال ۱۹۵۹ طراحی و ایجاد شد. کوبول بصورت یک زبان تجاری مورد استفاده قرار گرفت که نیاز به انجام عملیات دقیق بر روی مقادیر زیادی از داده‌ها دارد. امروزه در حدود نیمی از نرم‌افزارهای تجاری موجود توسط کوبول برنامه‌نویسی شده‌اند. به طور تقریبی در حدود یک میلیون نفر با این نرم‌افزار برنامه‌نویسی می‌کنند.

Pascal همزمان با C طراحی گردیده است. این زبان توسط پروفسور Nicklaus Wirth برای استفاده‌های آکادمیک ایجاد شده است.

۱-۱۲ بیسیک، ویژوال بیسیک، ویژوال ++C، #C و .NET

ایجاد و توسعه برنامه‌های کاربردی مبتنی بر ویندوز میکروسافت توسط زبان‌های هانند C و ++C کار مشکلی بوده و فرآیند پیچیده ای دارد. زمانیکه Bill Gates شرکت میکروسافت را تاسیس نمود، مبادرت به

[1] Personal Digital Assistant

پیاده‌سازی BASIC بر روی چندین کامپیوتر شخصی اولیه کرد. زبان BASIC[1] زبان برنامه‌نویسی است که در میانه دهه 1960 توسط پروفسور *John Kemeny* و *Thomas Kurtz* از کالج Dartmouth به منظور نوشتن برنامه‌های ساده ابداع گردید. هدف از BASIC اولیه، آموزش تکنیک‌های برنامه‌نویسی به افراد مبتدی و تازه کار بود.

با توسعه واسط گرافیکی کاربر (GUI) توسط میکروسافت، در اواخر دهه ۱۹۸۰ و اوایل ۱۹۹۰ بیسیک تکامل تدریجی خود را به سوی ویژوال بیسیک انجام داده بود که توسط گروه مایکروسافت در سال ۱۹۹۱ انجام پذیرفت.

اگرچه ویژوال بیسیک از زبان برنامه‌نویسی BASIC مشتق شده است، اما تفاوت‌های بسیار زیادی با BASIC دارد. ویژوال بیسیک از ویژگیهای قدرتمندی همانند واسط گرافیکی کاربر، رسیدگی به رویداد (*event handling*)، دسترسی به Win32 API، برنامه‌نویسی شی‌گرا، رسیدگی به خطا، برنامه‌نویسی ساخت‌یافته و سایر موارد برخوردار است. ویژوال بیسیک از جمله زبان‌های پر طرفدار در برنامه‌نویسی رویداد گرا (*event-driven*) و واسط‌های ویژوال است.

جدیدترین نسخه ویژوال بیسیک، ویژوال بیسیک NET. است، که برای پلات‌فرم جدید برنامه‌نویسی میکروسافت طراحی شده است. نسخه‌های اولیه ویژوال بیسیک دارای قابلیت شی‌گرا بودند، اما ویژوال بیسیک NET. در سطح بالاتری به عرضه برنامه‌نویسی شی‌گرا می‌پردازد و دارای کتابخانه NET. قدرتمندی از کامپونت‌های نرم‌افزاری با قابلیت استفاده مجدد است.

[نکته: خواننده عزیز، در صورتیکه به زبان برنامه‌نویسی ویژوال بیسیک NET. علاقمند باشید کتاب "راهنمای جامع برنامه نویسان ویژوال بیسک NET. ویرایش دوم" که توسط نویسنده‌گان همین کتاب نوشته شده، از سوی انتشارات اتحاد (نشر و پخش آیلار) ترجمه و چاپ شده است.]

ویژوال C++ نسخه پیاده سازی شده C++ توسط شرکت میکروسافت است. در اوایل، برنامه‌نویسی گرافیگی و GUI در ویژوال C++ با استفاده از *کلاس‌های بنیادین مایکروسافت*[2] (MFC) صورت می‌گرفت. در حال حاضر، با معرفی شدن NET. میکروسافت مبادرت به تهیه یک کتابخانه عمومی برای پیاده سازی GUI، گرافیک، شبکه، چندنخی و موارد دیگر کرده است. این کتابخانه مابین ویژوال بیسیک، ویژوال C++ و زبان برنامه‌نویسی جدید میکروسافت بنام C# مشترک است.

[1] Beginner's All-Purpose Symbolic Instruction Code
[2] Microsoft Foundation Classes

ابزارهای برنامه‌نویسی جدید (همانند C++ و جاوا) و دستگاه‌های الکترونیکی مصرف کننده (همانند تلفن‌های موبایل) مسائل و نیازهای خاص خود را دارند. جمع کردن کامپونت‌های نرم‌افزاری از زبانهای مختلف کار مشکلی است، و معمولا مشکل نصب در این بین رخ می‌دهد، چرا که نسخه‌های جدید از کامپونت‌های اشتراکی با نسخه‌های قدیمی سازگاری پیدا نمی‌کنند. علاوه بر این، توسعه دهنده‌گان متوجه نیازهای خود در زمینه برنامه‌های کاربردی مبتنی بر وب شدند که بتوان به آنها از طریق اینترنت دسترسی پیدا کرده و از آنها استفاده کرد. در نتیجه استقبال عمومی از دستگاه‌های الکترونیکی سیار، توسعه دهنده‌گان نرم‌افزار بر این باور رسیدند که مشتریان، دیگر نمی‌خواهند به کامپیوترهای رومیزی محدود باشند. توسعه دهنده‌گان نیاز خود به یک نرم‌افزار که بتواند در اختیار همه بوده و تقریبا برروی هر نوع دستگاه بکار گرفته شود، را احساس کردند. پاسخ میکروسافت به این نیازها NET. (با تلفظ دات نت) و زبان برنامه‌نویسی C# (با تلفظ سی شارپ) بود.

پلات‌فرم موجود در NET. یک مدل جدید از توسعه نرم‌افزاری عرضه می‌کند که به برنامه‌های ایجاد شده توسط زبان‌های مختلف برنامه‌نویسی امکان می‌دهد تا با یکدیگر ارتباط برقرار نمایند. زبان برنامه‌نویسی C# توسط میکروسافت و به سرپرستی *Anders Hejlsberg* و *Scott Wiltamuth* خاص پلات‌فرم NET. بعنوان زبانی که به برنامه نویسان اجازه می‌داد تا بتوانند به آسانی خود را به سطح NET. ارتقاء دهند، توسعه پیدا کرده است. با توجه به این حقیقت که ریشه C# از زبان‌های C, C++ و جاوا است، فرآیند ارتقاء به آسانی می‌تواند صورت گیرد. در حالیکه C# حاوی بهترین ویژگیهای زبان‌های فوق به همراه قابلیت‌های جدید متعلق به خود است. بدلیل اینکه در ساخت C# بطور گسترده‌ای از زبانهای مناسب و خوش ساختار استفاده شده است، برنامه نویسان از کار کردن و آسانی آن لذت می‌برند.

۱-۱۳ تکنولوژی شی

یکی از نویسنده‌گان این کتاب، پورفسور H.M.Deitel شاهد یک فروپاشی در اواخر دهه 1960 توسط سازمان‌های توسعه نرم‌افزار، بویژه در شرکتهای بود که پروژه‌هائی در سطح کلان انجام می‌دادند. زمانیکه Deitel دانشجو بود، فرصت کار کردن در یک گروه توسعه دهنده در بخش اشتراک زمانی و حافظه مجازی سیستم‌های عامل را بدست آورد، که در نوع خود یک تجربه با ارزش محسوب می‌شد. اما در تابستان 1967، شرکت تصمیم به متوقف کردن پروژه بدلایل اقتصادی کرد، پروژهای که صدها نفر بمدت چندین سال بر روی آن کار کرده بودند. این موضوع نشان داد که نرم‌افزار محصول پیچیده‌ای است.

بهبود وضعیت نرم‌افزار از زمانی آغاز شد که مزایای برنامه‌نویسی *ساخت‌یافته* بر همگان هویدا گردید (از دهه 970). اما تا دهه 1990 تکنولوژی برنامه‌نویسی شی‌گرا بطرز گسترده‌ای بکار گرفته نشد، تا اینکه برنامه‌نویسان به وجود ابزاری برای بهبود فرآیند توسعه نرم‌افزار، احساس نیاز کردند.

در واقع، پیدایش تکنولوژی شی به میانه دهه 1960 باز می‌گردد. زبان برنامه‌نویسی C++ در AT&T و توسط Bjarne Stroustrup در اوایل دهه 1980 توسعه یافته که خود مبتنی بر دو زبان C و Simula 67 است. زبان C در بدو کار در AT&T و برای پیاده سازی سیستم عامل یونیکس در اوایل دهه 1970 ایجاد شد. زبان Simula 67، زبان برنامه‌نویسی شبیه سازی است که در سال 1967 و در اروپا توسعه یافته است. زبان C++ در برگیرنده قابلیت‌ها و توانائی زبان‌های C و Simula در ایجاد و دستکاری شی‌ها است.

شی‌ها چیستند و چرا خاص می‌باشند؟ در واقع، تکنولوژی شی، یک الگوی بسته (package) است که به ما در ایجاد واحدهای نرم‌افزاری با معنی کمک می‌کند. شی‌ها تمرکز زیادی بر نواحی خاص برنامه‌ها دارند. شی‌ها می‌توانند از جمله شی‌های تاریخ، زمان، پرداخت، فاکتور، صدا، ویدئو، فایل، رکورد و بسیاری از موارد دیگر باشد. در حقیقت، می‌توان هر چیزی را بفرم یک شی عرضه کرد.

ما در دنیای از شی‌ها زندگی می‌کنیم. کافیست نگاهی به اطراف خود بیاندازیم. اطراف ما پر است از اتومبیل‌ها، هواپیما، انسان‌ها، حیوانات، ساختمان‌ها، چراغ‌های ترافیگ، بالابرها، و بسیاری از چیزهای دیگر. قبل از اینکه زبان‌های برنامه‌نویسی شی‌گرا ابداع شوند، زبان‌های برنامه‌نویسی (همانند FORTRAN، Pascal ، C و Basic) بر روی اعمال یا actions بجای چیزها یا شی‌ها (نام) تمرکز داشتند. با اینکه برنامه‌نویسان در دنیائی از شی‌ها زندگی می‌کردند اما با فعل‌ها سرگرم بودند. خود همین تناقض باعث شد تا برنامه‌های نوشته شده از قدرت کافی برخوردار نباشند. هم اکنون که زبان‌های برنامه‌نویسی شی‌گرا همانند C و جاوا در دسترس هستند، برنامه‌نویسان به زندگی خود در یک دنیای شی‌گرا ادامه می‌دهند و می‌توانند برنامه‌های خود را با اسلوب شی‌گرا بنویسند. فرآیند برنامه‌نویسی شی‌گرا در مقایسه با برنامه‌نویسی *روالی* (*procedural*) ماهیت بسیار طبیعی‌تری دارد و نتیجه آن هم رضایت بخش‌تر است.

اصلی‌ترین مشکل برنامه‌نویسی روالی این است که واحدهای تشکیل دهنده برنامه نمی‌توانند به آسانی نشاندهنده موجودیت‌های دنیای واقعی باشند. از اینرو، این واحدها نمی‌توانند بطرز شایسته‌ای از قابلیت استفاده مجدد برخوردار باشند. برای برنامه نویسان روالی، نوشتن دوباره و مجدد کدها در هر پروژه جدید و با وجود مشابه بودن این کدها با نرم‌افزار پروژه‌های قبلی، چندان غیر عادی نمی‌باشد. نتیجه اینکار تلف شدن زمان و هزینه‌ها است. با تکنولوژی شی، موجودیت‌های نرم‌افزاری (کلاس) ایجاد می‌شوند و در صورتیکه بدرستی طراحی شده باشند، می‌توان در آینده از آنها در پروژه‌های دیگر استفاده کرد.

استفاده از کتابخانه کامپونت‌ها با قابلیت بکارگیری مجدد همانند *MFC(Microsoft Foundation Classes)* و کامپونت‌های تولید شده توسط *Rogue Wave* و سایر شرکتهای نرم‌افزاری، می‌تواند در کاهش هزینه‌ها و نیازهای پیاده‌سازی سیستم‌های خاص بسیار موثر باشد.

با بکارگیری برنامه‌نویسی شی‌گرا، نرم‌افزار تولید شده بسیار قابل فهم‌تر شده، نگهداری و سازماندهی آن اصولی‌تر و اصلاح و خطایابی آن ساده‌تر می‌شود. این موارد از اهمیت خاصی برخوردار هستند چراکه تخمین زده می‌شود که هشتاد در صد هزینه یک نرم‌افزار مربوط به دوره نگهداری و ارتقاء آن در چرخه طول عمرش است و ارتباطی با نوشتن و توسعه اولیه نرم‌افزار ندارد. با تمام این اوصاف، مشخص است که برنامه‌نویسی شی‌گرا تبدیل به یکی از کلیدی‌ترین مفاهیم برنامه‌نویسی در چند دهه آینده خواهد شد.

یکی از دلایل نوشتن کدهای متعلق بخود این است که دقیقا می‌دانید که کد نوشته شده چگونه کار می‌کند، و می‌توانید به بررسی آن بپردازید. عیب اینکار هم در صرف زمان و پیچیدگی طراحی و پیاده‌سازی کد جدید است.

مهندسی نرم‌افزار

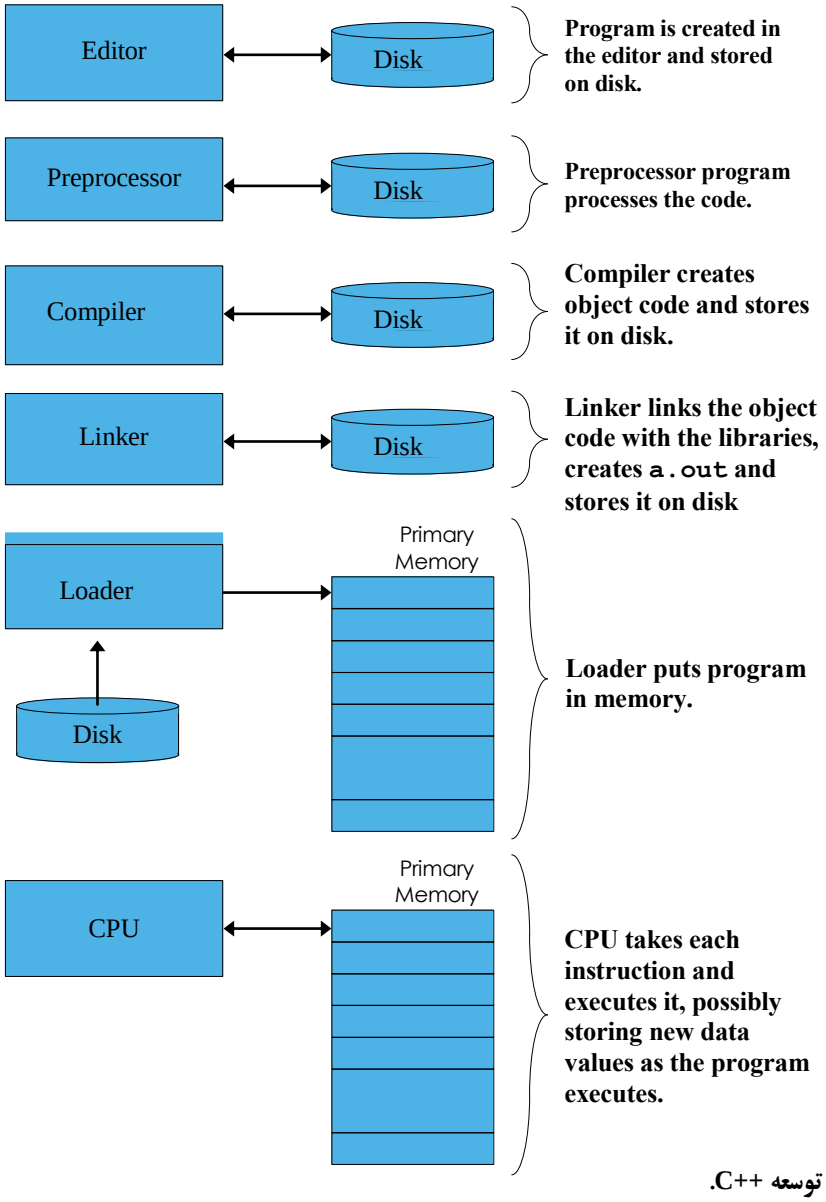
کتابخانه کامپونت‌های نرم‌افزاری با قابلیت استفاده مجدد را می‌توان از طریق اینترنت و وب گسترده جهانی بدست آورد. برای تهیه بسیاری از این کتابخانه‌ها نیازی به پرداخت هزینه نیست.

۱۴-۱ محیط توسعه C++

اجازه دهید تا به بررسی مراحل ایجاد و اجرای یک برنامه کاربردی C++ با استفاده از محیط توسعه C++ بپردازیم (شکل ۱-۱). اصولاً سیستم C++ متشکل از سه بخش است: *محیط توسعه برنامه*, *زبان* و *کتابخانه استاندارد* C++. عموماً برنامه‌های C++ از شش فاز یا مرحله عبور می‌کنند: ویرایش, پیش پردازش، کامپایل, لینک, بار شدن و اجرا. در ادامه به توضیح هر فاز می‌پردازیم.

فاز ۱: ایجاد برنامه

فاز ۱ مرکب از ویرایش یک فایل با استفاده از یک برنامه ویرایشگر است. با استفاده از یک ویرایشگر مبادرت به تایپ برنامه C++ کرده و هرگونه اصلاحات مورد نیاز را در آن اعمال و بر روی یک دستگاه ذخیره‌سازی ثانویه, همانند دیسک سخت ذخیره می‌کنیم. فایل‌های برنامه C++ دارای پسوندهای ***.cpp***, ***.cxx*** , ***.cc*** یا ***.C***. می‌باشند (دقت کنید که C با حرف بزرگ است), که نشان می‌دهند, که فایل حاوی کد منبع C++ است. برای مشاهده پسوند فایل در محیط توسعه C++ به مستندات محیط توسعه C++ بکار رفته بر روی سیستم خود مراجعه کنید.



شکل ۱-۱ | محیط توسعه C++.

دو ویرایشگر که در سیستم UNIX کاربرد بیشتری دارند عبارتند از **vi** و **emacs**. بسته‌های نرم‌افزاری C++ برای ویندوز میکروسافت همانند

Metroworks CodeWarrior (www.metrowerks. com), Borland C++ (www.borland. com)

و Microsoft Visual C++ (www.msdn.microsoft.com/visualc/) دارای ویرایشگر مجتمع در محیط برنامه‌نویسی خود هستند. البته می‌توانید از یک ویرایشگر متنی ساده همانند Notepad در ویندوز, برای نوشتن کد C++ استفاده کنید. فرض ما بر این است که شما حداقل با نحوه ویرایش یک برنامه آشنا هستید.

فاز ۲ و ۳: پیش پردازنده و کامپایل یک برنامه C++

در فاز ۲, برنامه‌نویس, دستوری برای کامپایل برنامه صادر می‌کند. در یک سیستم C++, برنامه *پیش‌پردازنده*, قبل از اینکه فاز ترجمه کامپایلر شروع بکار کند, آغاز یا اجرا می‌شود (از اینرو به پیش‌پردازنده فاز ۲ و به کامپایل فاز ۳ نام گذاشته‌ایم). پیش‌پردازنده فرمانبر دستوراتی بنام *رهنمود پیش‌پردازنده* است که عملیات مشخص بر روی برنامه قبل از کامپایل آن انجام می‌دهند. معمولاً این عملیات شامل کامپایل سایر فایلهای متنی و انجام انواع جایگزینی‌ها است. در فصل‌های آتی با تعدادی از رهنمودهای پیش‌پردازنده آشنا خواهید شد. در فاز ۳, کامپایلر شروع به ترجمه برنامه C++ به کد زبان ماشین می‌کند که گاهاً بعنوان *کد شی* (*object code*) شناخته می‌شود.

فاز ٤: لینک

فاز ۴ معروف به فاز لینک است. معمولاً برنامه‌های C++ حاوی مراجعه‌هایی به توابع و داده‌های تعریف شده در بخش‌های گوناگون هستند, همانند کتابخانه‌های استاندارد یا کتابخانه‌های خصوصی گروهی از برنامه‌نویسان که بر روی یک پروژه کار می‌کنند. کد شی تولید شده توسط کامپایلر C++ حاوی شکاف‌هایی است که از فقدان این بخش‌ها بوجود می‌آیند. برنامه *لینکر* مبادرت به لینک کد شی با کد توابع مفقوده می‌کند تا یک تصویر کامل و بدون شکاف بوجود آید. اگر برنامه بدرستی کامپایل و لینک گردد, یک نماد اجرایی تولید خواهد شد.

فاز ٥: بار شدن

فاز ۵, فاز *بار* شدن نام دارد. قبل از اینکه برنامه بتواند اجرا شود, ابتدا بایستی در حافظه جای گیرد. این کار توسط *بارکننده* (*loader*) صورت می‌گیرد که نماد اجرایی را از دیسک دریافت و به حافظه منتقل می‌کند. همچنین برخی از کامپونت‌ها که از برنامه پشتیبانی می‌کنند, به حافظه بار می‌شوند.

فاز ٦: اجرا

سرانجام, کامپیوتر, تحت کنترل CPU خود, مبادرت به اجرای یک به یک دستورالعمل‌های برنامه می‌کند.

مسائلی که ممکن است در زمان اجرا رخ دهند

همیشه برنامه‌ها در همان بار اول اجرا نمی‌شوند. در هر کدامیک از فازهای فوق امکان شکست وجود دارد، چرا که ممکن است انواع مختلفی از خطاها که در مورد آنها صحبت خواهیم کرد, در این بین رخ دهند. برای مثال, امکان دارد برنامه اقدام به انجام عملیات تقسیم بر صفر نماید (یک عملیات غیرمعتبر). در

چنین حالتی برنامه ++C یک پیغام خطا بنمایش در خواهد آورد. اگر چنین شود, مجبور هستید تا به فاز ویرایش بازگردید و اصلاحات لازم را انجام داده و مجدداً مابقی فازها را برای تعیین اینکه آیا اصلاحات صورت گرفته قادر به رفع مشکل بوده‌اند یا خیر, طی کنید.

بیشتر برنامه‌های نوشته شده در ++C عملیات ورود و یا خروج داده انجام می‌دهند. توابع مشخصی از ++C, از طریق **cin** (استریم ورودی استاندارد, با تلفظ "see-in") که معمولاً از طریق صفحه‌کلید است, اقدام به دریافت ورودی می‌کنند, اما تابع **cin** می‌تواند به دستگاه دیگری هدایت شود. غالباً داده‌ها از طریق دستور **cout** (استریم استاندارد خروجی, با تلفظ "see-out") خارج می‌گردند که معمولاً صفحه‌نمایش کامپیوتر است, اما می‌توان **cout** را به دستگاه دیگری هدایت کرد. زمانیکه می‌گوییم برنامه نتیجه‌ای را چاپ می‌کند, معمولاً منظورمان نمایش نتیجه بر روی صفحه نمایش (مانیتور) است. می‌توان داده‌ها را به دستگاه‌های دیگری همانند دیسک‌ها ارسال کرد و توسط چاپگر از آنها چاپ گرفت. همچنین دستوری بنام **cerr** (استریم استاندارد خطا) وجود دارد, که از آن برای نمایش پیغام‌های خطا استفاده می‌شود, این دستور اکثراً برای نمایش پیغام در صفحه نمایش بکار گرفته می‌شود.

خطای برنامه‌نویسی

خطاهایی همانند, خطای تقسیم بر صفر در زمان اجرای برنامه رخ می‌دهند, از اینرو به آنها خطای زمان اجرا گفته می‌شود. خطاهای زمان اجرای عظیم یا مهلک (fatal runtime errors) سبب می‌شوند تا برنامه‌ها بلافاصله خاتمه یابند بدون اینکه بتوانند با موفقیت وظایف خود را به انجام برسانند. خطاهای زمان اجرای غیرمهلک اجازه می‌دهند تا برنامه‌ها بطور کامل اجرا شوند, اما نتایج اشتباهی تولید می‌کنند.

۱۵-۱ نکاتی در مورد ++C و این کتاب

گاهی اوقات برنامه‌نویسان با تجربه ++C بدون رعایت هیچ‌گونه ساختار و روش مشخص شروع به برنامه‌نویسی می‌کنند, که چنین کاری در برنامه‌نویسی روش ضعیفی است. خواندن, نگهداری, تست و خطایابی چنین برنامه‌هایی بسیار مشکل بوده و گاهاً رفتار عجیبی از خود بروز می‌دهند. این کتاب با در نظر گرفتن برنامه‌نویسان مبتدی سعی در نوشتن واضح برنامه‌ها دارد.

برنامه‌نویسی ایده‌ال

برنامه‌های ++C خود را ساده و سرراست بنویسید. گاهی اوقات به این روش KIS (keep it simple) گفته می‌شود. می‌دانید که C و ++C از جمله زبان‌های قابل حمل می‌باشند و برنامه‌های نوشته شده در این دو زبان قادر به اجرا برای انواع کامپیوترها هستند. قابلیت حمل یک بحث اغواکننده است. مستند ANSIC استاندارد حاوی لیست طولانی از مباحث قابلیت حمل است و کتاب‌های کاملی در این زمینه نوشته شده‌اند.

قابلیت حمل

اگر چه نوشتن برنامه‌های قابل حمل امکان‌پذیر است, چندین مسئله در میان انواع مختلف کامپایلرهای C و

++C و کامپیوترها وجود دارد که می‌توانند قابلیت حمل را مشکل‌ساز کنند. نوشتن برنامه‌ها در C و ++C قابلیت حمل را تضمین نمی‌کند. غالباً برنامه‌نویس نیاز به توجه دقیق به نوع کامپایلر و کامپیوتر دارد، که بصورت سرجمع پلات فرم (platform) نام دارد.

مطالب عرضه شده در این کتاب براساس مستندات استاندارد ANSI/ISO C++ نوشته شده‌اند. با این همه, C++ یک زبان غنی است و برخی از ویژگی‌های آن در این کتاب مورد بررسی قرار نگرفته‌اند. اگر نیاز به جزئیات تکنیکی دیگری در ارتباط با C++ داشته باشید, بایستی به مطالعه مستندات استاندارد C++ بپردازید، که می‌توان از وب سایت ANSI به آن دسترسی پیدا کرد.

webstore.ansi.org/ansidocstore/default.asp

مستند عنوان "Information Technology-Programming Language-C++" داشته و شماره مستند INCITS/ISO/IEC 14882-2003 است.

۱-۱۶ تست یک برنامه C++

در این بخش, مبادرت به اجرا و تعامل با اولین برنامه C++ خود می‌کنیم. کار را با اجرای *بازی حدس عدد* شروع می‌کنیم که عددی از 1 تا 1000 دریافت و شروع به حدس عدد می‌کند. اگر عددی که وارد کرده‌اید یا حدس زده‌اید, صحیح باشد, بازی به پایان رسیده است. اگر حدس شما صحیح نباشد, برنامه نشان خواهد داد که عدد وارد شده بزرگتر یا کوچکتر از عدد صحیح است. برای حدس زدن عدد هیچ محدودیتی در تعداد حدس وجود ندارد.

اجرای یک برنامه C++ را به دو روش به شما نشان خواهیم داد, استفاده از ***خط اعلان*** یا ***دستور*** (***Command Prompt***) ویندوز XP و استفاده از یک ***پوسته*** (***shell***) در لینوکس. اجرای برنامه در هر دو پلات‌فرم یکسان است. محیط‌های توسعه متعددی وجود دارند که شما بعنوان خواننده این کتاب می‌توانید با استفاده از آنها اقدام به کامپایل, ایجاد و اجرای برنامه‌های C++ کنید, محیط‌های همانند Borland C++ Builder, Metrowerks, GNU C++, Microsoft Visual C++. NET و غیره در حالیکه ما تمام این محیط‌ها را تست نکرده‌ایم, اما اطلاعاتی در بخش ۱۹-۱ فراهم آورده‌ایم که در ارتباط با کامپایلرهای مجانی C++ موجود در اینترنت هستند و می‌توان آنها را برداشت کرد. لطفاً با توجه به دستورالعمل مدرس خود اقدام به تهیه محیط توسعه کنید.

با توجه به مراحل معرفی شده در زیر, قادر به اجرای برنامه و وارد کردن اعداد مختلف برای حدث عدد صحیح خواهید بود. عناصر و مراحلی که در این برنامه مشاهده می‌کنید در کل این کتاب کاربرد خواهند داشت.

اجرای یک برنامه C++از طریق خط دستور ویندوز XP

۱- *بررسی setup*. برای اینکه مطمئن شوید مثال‌های کتاب به درستی بر روی دیسک‌سخت کامپیوترتان کپی شده‌اند, به مطالعه بخش «قبل از شروع بکار» در ابتدای کتاب کنید.

۲- *یافتن برنامه کامل*. پنجره خط دستور را باز کنید. برای کاربران در حال استفاده از ویندوز 95, 98 یا 2000, این پنجره از طریق ***Start>Programs>Accessories>Command Prompt*** باز می‌شود. برای کاربران ویندوز XP, این پنجره از طریق ***Start>All Programs>Accessories>Command Prompt*** باز می‌شود. برای رفتن به سراغ شاخه برنامه **GussNumber**, مبادرت به تایپ ***cd C:\examples\ch01\GussNumber\windows*** کرده سپس کلید *Enter* را فشار دهید (شکل ۲-۱). از دستور **cd** برای تغییر شاخه استفاده می‌شود.

۳- *اجرای برنامه **GuessNumber***. اکنون که در شاخه حاوی برنامه **GuessNumber** هستید, دستور **GuessNumber** را تایپ و کلید *Enter* را فشار دهید (شکل ۳-۱). [نکته: **GuessNumber.exe** نام واقعی برنامه است. با این وجود, فرض ویندوز پسوند **exe.** است.]

۴- *وارد کردن اولین حدس*. برنامه پیغام "`Please type first guess.`" را بنمایش در آورده و سپس یک علامت سئوال (?) در خط بعدی ظاهر می‌سازد (شکل ۳-۱). در این خط عدد 500 را وارد کنید (شکل ۴-۱).

۵- *وارد کردن حدس بعدی*. برنامه پیغام "`Too high. Try again`" را به نمایش در می‌آورد, به این معنی که مقدار وارد شده بزرگتر از عدد انتخابی از سوی برنامه به عنوان مقدار صحیح است. از اینرو, باید یک عدد کوچکتر برای حدس بعدی خود وارد کنید. در خط اعلان, مقدار 250 را وارد سازید (شکل ۵-۱). برنامه مجدداً پیغام "`Too high. Try again`" را بنمایش در می‌آورد, چرا که مقدار وارد شده هنوز بزرگتر از عدد صحیح مورد نظر می‌باشد.

شکل ۲-۱ | باز کردن پنجره Command Prompt و تغییر شاخه.

شکل ۳-۱ | اجرای برنامه GuessNumber

شکل ٤-۱ | وارد کردن اولین حدس.

شکل ٥-۱ | وارد کردن دومین حدس و پاسخ دریافتی.

۶- *وارد کردن حدس‌های بعدی*. با وارد کردن مقادیری به بازی ادامه دهید تا موفق به حدس عدد صحیح شوید. پس از حدس پاسخ صحیح, برنامه پیغام "`Excellent! You guessed the number!`" را بنمایش در می‌آورد (شکل ۶-۱).

۷- *بازی مجدد یا خروج از برنامه*. پس از حدس عدد صحیح, برنامه در مورد انجام یک بازی دیگر سئوال می‌کند (شکل ۶-۱). در مقابل پیغام "**Would you like to play again (y or n)?**" وارد کردن کاراکتر **y** سبب می‌شود تا برنامه یک عدد جدید انتخاب کرده و مجدداً با نمایش پیغام "**Please enter your first guess.**" و بدنبال آن یک علامت سئوال در خط بعدی یک بازی جدید را شروع کند (شکل ۷-۱). با وارد کردن کاراکتر **n** برنامه خاتمه یافته و به شاخه برنامه در خط دستور باز می‌گردد (شکل ۸-۱). در هر بار اجرای این برنامه از مرحله ۳, همان اعداد برای حدس انتخاب می‌شوند.

۸- *بستن پنجره* ***Command Prompt***

شکل ٦-١ | وارد کردن اعداد دیگر و حدس عدد صحیح.

شکل ۷-۱ | بازی مجدد.

شکل ۸-۱ | خروج از بازی.

اجرای یک برنامه ++C با استفاده از GNU C++ در لینوکس

برای تست این برنامه، فرض می‌کنیم که شما با نحوه کپی مثال‌ها به شاخه خانگی خود آشنا هستید. اگر سئوالی در این زمینه دارید، می‌توانید از مدرس خود کمک بگیرید. خط اعلان در پوسته سیستم از کاراکتر مد (~) استفاده می‌کند تا نشاندهنده شاخه خانگی (home) باشد و هر اعلان با کاراکتر **$** پایان می‌یابد. خط اعلان در سیستم‌های لینوکس بسیار متفاوت است.

۱- *یافتن برنامه*. از طریق یک پوسته لینوکس، به شاخه برنامه **GuessNumber** بروید (شکل ۹-۱)، با تایپ

```
cd Examples\ch01\GuessNumber\GNU_Linux
```

سپس کلید *Enter* را فشار دهید. دستور **cd** برای تغییر شاخه بکار گرفته می‌شود.

۲- *کامپایل برنامه* **GuessNumber**. برای اجرای برنامه در کامپایلر GNU C++، بایستی ابتدا آن را توسط عبارت زیر کامپایل کنید

```
g++ GuessNumber.cpp -o GuessNumber
```

همانند شکل ۱۰-۱. دستور فوق سبب کامپایل برنامه و تولید یک فایل اجرایی بنام **GuessNumber** در خط اعلان بعدی می‌شود. کلید *Enter* را فشار دهید (شکل ۱۱-۱).

```
~$ cd examples/ch01/GuessNumber/GNU_linux
~/examples/ch01/GuessNumber/GNU_Linux$
```

شکل ۹-۱ | تغییر شاخه برنامه GussNumber پس از ورود به سیستم لینوکس.

```
~/examples/ch01/GuessNumber/GNU_linux$ g++ GuessNumber.cpp -o GuessNumber
```

```
~/examples/ch01/GuessNumber/GNU_linux$
```

شکل ۱۰-۱ | کامپایل برنامه GuessNumber با استفاده از دستور g++.

```
~/examples/ch01/GuessNumber/GNU_linux$  ./GuessNumber
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
?
```

شکل ۱۱-۱ | اجرای برنامه GuessNumber

۴- *وارد کردن حدس اول.* برنامه عبارت "**Pleas type your first guess.**" و سپس نماد علامت سئوال (**?**) را بعنوان اعلان در خط بعدی بنمایش در می‌آورد (شکل ۱۱-۱). در این اعلان عدد 500 را وارد سازید (شکل ۱۲-۱).

۵- *وارد کردن حدس بعدی.* "**Too high. Try again.**" را بنمایش در می‌آورد، به این معنی که مقدار ورودی بزرگتر از عدد انتخابی از سوی برنامه است (شکل ۱۲-۱). در اعلان بعدی، مقدار 250 را وارد کنید (شکل ۱۳-۱). این بار برنامه پیغام "**Too low. Try again**" را نشان می‌دهد. چرا که مقدار وارد شده کوچکتر از مقدار صحیح است.

۶- *وارد کردن حدس‌های بعدی.* به بازی با وارد کردن حدس‌های بعدی ادامه دهید تا عدد صحیح را حدس بزنید. پس از حدس زدن پاسخ، "**Excellent! You guessed the number!**" بنمایش در می‌آید (شکل ۱۴-۱).

```
~/examples/ch01/GuessNumber/GNU_linux$  ./GuessNumber
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too hight.Try again.
?
```

شکل ۱۲-۱ | وارد کردن حدس اول.

```
~/examples/ch01/GuessNumber/GNU_linux$  ./GuessNumber
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too hight.Try again.
? 250

Too low. Try again.

?
```

شکل ۱۳-۱ | وارد کردن حدس دوم و دریافت پاسخ برنامه.

```
Too low. Try again.
?  375
Too low. Try again.
?  437
Too high. Try again.
?  406
```

```
Too high. Try again.
? 391
Too low. Try again.
? 387
Too high. Try again.
? 385
Too high. Try again.
? 384

Excellent! You guessed the number!
Would you like to play again <y or n>?
```

شکل ١٤-١ | وارد کردن حدس‌های بعدی و حدس پاسخ صحیح.

۷- *بازی مجدد یا خروج از برنامه*. پس از حدس عدد صحیح، برنامه در مورد انجام یک بازی دیگر سئوال می‌کند. در مقابل پیغام "**Would you like to play again (y or n)?**" وارد کردن کاراکتر **y** سبب می‌شود تا برنامه یک عدد جدید انتخاب کرده و مجدداً با نمایش "**Please enter your first guess.**" و بدنبال آن یک علامت سئوال در خط بعدی یک بازی جدید را شروع کند (شکل ۱۵-۱). با وارد کردن کاراکتر **n** برنامه خاتمه یافته و به شاخه برنامه در پوسته باز می‌گردید (شکل ۱۶-۱). در هر بار اجرای این برنامه از مرحله ۳، همان اعداد برای حدس انتخاب می‌شوند.

```
Excellent! You guessed the number!
Would you like to play again <y or n>? y

I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
?
```

شکل ۱۵-۱ | بازی مجدد.

```
Excellent! You guessed the number.
Would you like to play again <y or n>? n
~/examples/ch01/GuessNumber/GNU_linux$
```

شکل ١٦-١ | خروج از بازی.

۱-۱۷ مبحث آموزشی مهندسی نرم‌افزار: مقدمه‌ای بر تکنولوژی شی و UML

اکنون بحث خود را در ارتباط با شی‌گرا و تفکر طبیعی در ارتباط با جهان و برنامه‌نویسی کامپیوتر آغاز می‌کنیم. در انتهای تمام فصل‌های ۱ الی ۷ و ۱۳ بخشی بعنوان مبحث آموزشی مهندسی نرم‌افزار وجود دارد که در آن به معرفی دقیق بحث شی‌گرا می‌پردازیم. هدف ما از این بخش کمک به شما در توسعه یک روش تفکر شی‌گرا و آشنایی با ***UML (Unified Modeling Language)*** است. یک زبان گرافیکی که به افراد مسئول طراحی سیستم‌های نرم‌افزاری شی‌گرا امکان می‌دهد تا از نمادهای استاندارد شده صنعتی برای نمایش طرح‌ها استفاده کنند. در این بخش که مطالعه آن الزامی است، به معرفی مفاهیم پایه شی‌گرا و اصطلاحات فنی آن خواهیم پرداخت. بخش‌های اختیاری قرار گرفته در فصل‌های ۲ الی ۷، ۹ و ۱۳ نمایشی از یک طراحی شی‌گرا و پیاده‌سازی نرم‌افزاری برای یک ماشین خودپرداز (ATM) ساده ارائه می‌کنند.

بخش‌های مهندسی نرم‌افزار در انتهای فصل‌های ۲ الی ۷ عبارتند از:

- تحلیل نیازمندی‌های یک سیستم نرم‌افزاری (سیستم ATM) برای ایجاد آن
- تعیین صفاتی که شی‌ها باید داشته باشند
- تعیین رفتار این شی‌ها
- تعیین نحوه تعامل شی‌ها با یکدیگر برای برطرف کردن نیازهای سیستم

مفاهیم پایه تکنولوژی شی

مقدمه بحث شی‌گرا را با برخی اصطلاحات کلیدی آغاز می‌کنیم. به هر کجا در دنیای واقعی نگاه کنید. شاهد *شی‌ها* خواهید بود. مردم، حیوانات، گیاهان، اتومبیل‌ها، هواپیماها، ساختمان‌ها، کامپیوترها و بسیاری از چیزهای دیگر. تفکر انسان براساس شی است. تلفن‌ها، خانه‌ها، چراغ‌های ترافیک، اجاق‌های میکروویو و کولرهای آبی تعدادی کمی از بی‌شمار شی‌ها هستند که در زندگی روزمره خود شاهد آنها هستیم.

گاهی اوقات مبادرت به تقسیم شی‌ها به دو طبقه می‌کنیم: متحرک و غیرمتحرک. شی‌های متحرک، زنده هستند و می‌توانند حرکت کرده و کاری انجام دهند. از سوی دیگر، شی‌های غیرمتحرک، در محیط خود حرکت نمی‌کنند. با این همه، شی‌ها از هر دو نوع، در برخی چیزها مشترک هستند. همه آنها دارای *صفات* (همانند سایز، شکل، رنگ و وزن) بوده و از خود *رفتارهای* نشان می‌دهند (مثلاً توپ می‌غلتد، بالا و پایین می‌پرد، یک بچه گربه می‌کند، می‌خوابد، راه می‌رود، یک اتومبیل شتاب می‌گیرد، ترمز کرده و چرخش می‌کند، یک حوله آب را جذب می‌کند).

انسان‌ها با بررسی صفات و رفتار اشیاء، مطالبی در ارتباط با آنها یاد می‌گیرند. شی‌های مختلف می‌توانند صفات و رفتار مشابهی داشته باشند. مقایسه را همیشه می‌توان اعمال کرد، مثلاً مابین کودکان و بزرگسالان.

طراحی نرم‌افزار مدل‌های شی‌گرا ***OOD (Object-oriented design)*** شبیه به روشی است که افراد برای توصیف شی‌ها در دنیای واقعی بکار می‌برند. می‌توان از رابطه موجود مابین کلاس استفاده کرد، آنجا که شی‌هایی از یک کلاس همانند کلاس وسیله نقلیه دارای ویژگی‌های یکسان هستند، اتومبیل‌ها، تریلی‌ها دارای صفات مشترک می‌باشند. OOD از مزیت *ارث‌بری* (یا توارث) سود می‌برد، که در آن کلاس‌های جدید، صفاتی را از کلاس‌های موجود جذب می‌کنند و به صفات خاص خود اضافه می‌نمایند.

طراحی شی‌گرا یک روش طبیعی و ذاتی در نمایش فرآیند طراحی نرم‌افزار بنام مدلسازی شی‌ها توسط صفات، رفتار و رابطه موجود در میان شی‌ها همانند شی‌ها در دنیای واقعی است. همچنین OOD ارتباط مابین شی‌ها را مدلسازی می‌کند. همانند فردی که پیغامی به شخص دیگری می‌فرستد، شی‌ها نیز از طریق پیغام ارتباط برقرار می‌کنند. یک شی حساب بانکی می‌تواند پیغامی برای کاهش موجودی دریافت کند چرا که مشتری مقداری از پول خود را از حساب برداشت کرده است.

OOD مبادرت به *کپسوله‌سازی* صفات و *عملیات* (رفتار) در شی‌ها می‌کند. سرانجام صفات و رفتار یک شی با هم گره می‌خورند. شی‌ها از خصیصه *پنهان‌سازی اطلاعات* برخوردار هستند. به این معنی که شی‌ها از نحوه برقراری ارتباط با یک شی دیگر از طریق یک *واسط* (رابط) مناسب مطلع هستند، اما معمولاً از نحوه پیاده‌سازی شی‌های دیگر اطلاعی ندارند. معمولاً جزئیات پیاده‌سازی در درون خود شی‌ها پنهان می‌باشند. می‌توان این امر را با رانندگی اتومبیل مقایسه کرد که در آن برای راندن اتومبیل نیازی نیست تا با جزئیات موتور، جعبه‌دنده و عملکرد ترمز آشنا بود. در ادامه با نحوه پنهان‌سازی اطلاعات و دلایل آن آشنا خواهید شد، که از جمله موارد مناسب در مهندسی نرم‌افزار است.

زبان‌های همانند C++ *شی‌گرا* هستند. به برنامه‌نویسی در چنین زبان‌های، *برنامه‌نویسی شی‌گرا* (*OOP*) گفته می‌شود و به برنامه‌نویسان کامپیوتری امکان می‌دهند تا یک طرح شی‌گرا را بصورت یک سیستم نرم‌افزاری پیاده‌سازی کنند. از سوی دیگر زبان‌های همانند C، زبان‌های *روالی* هستند و از اینرو برنامه‌نویسان مقید به *عمل* می‌باشند. در C واحد برنامه‌نویسی *تابع* است، در حالیکه در C++ این واحد، *کلاس* می‌باشد. کلاس‌های C++ حاوی توابعی هستند که پیاده‌سازی‌کننده صفات می‌باشند.

برنامه‌نویسان C بر روی نوشتن توابع تمرکز دارند. آنها گروهی از عمل‌ها را که وظایفی انجام می‌دهند در درون یک تابع قرار می‌دهند و سپس گروهی از توابع را به فرم یک برنامه در می‌آورند. بطور مشخص داده‌ها در C بسیار با ارزش هستند، اما از همان ابتدای امر برای پشتیبانی از اعمالی که توابع انجام خواهند داد وجود دارند.

کلاس‌ها، اعضا داده و توابع عضو

تمرکز برنامه‌نویسان C++ بر روی ایجاد «*نوع‌های تعریف شده از سوی کاربر*» که *کلاس* نامیده می‌شوند است. هر کلاس حاوی داده و هم مجموعه‌ای از متدها است که بر روی داده‌ها کار می‌کنند و سرویس‌های برای *سرویس‌گیرنده‌ها* (کلاس‌ها و توابع دیگری که از کلاس استفاده می‌کنند) تدارک می‌بینند، به کامپونت‌های داده از یک کلاس *اعضای داده* گفته می‌شود. برای مثال، یک کلاس حساب بانکی می‌تواند شامل یک شماره حساب و یک موجودی باشد. به کامپونت‌های داده از یک کلاس، *توابع عضو* گفته می‌شود (معمولاً در سایر زبان‌های برنامه‌نویسی شی‌گرا همانند جاوا، به توابع عضو، *متد* گفته می‌شود). برای مثال، یک کلاس حساب بانکی می‌تواند دارای توابع عضوی برای ایجاد یک پس‌انداز (افزایش‌دهنده موجودی)، برداشت (کاهش‌دهنده موجودی) و نمایش موجودی فعلی باشد. برنامه‌نویس از نوع‌های توکار (و سایر نوع‌های تعریف شده توسط کاربر) بعنوان «بلوک‌های سازنده» در ایجاد نوع‌های جدید (کلاس‌ها) استفاده می‌کند. اسامی در ساختار یک سیستم به برنامه‌نویس C++ در تعیین اینکه کدام شی‌ها برای کار با یکدیگر طراحی شده‌آند کمک می‌کنند.

کلاس‌ها همانند نقشه ترسیمی خانه‌ها هستند. یک کلاس، نقشه ایجاد یک شی از کلاس است. همانطوری که می‌توانیم خانه‌های متعددی از روی یک نقشه بسازیم، می‌توانیم تعدادی شی از روی یک کلاس، نمونه‌سازی کنیم. نمی‌توان در نقشه آشپزخانه مبادرت به آشپزی کرد، آشپزی فقط در آشپزخانه خانه امکان‌پذیر است.

کلاس‌ها می‌توانند با کلاس‌های دیگر ارتباط داشته باشند. برای مثال در یک طرح شی‌گرا از بانک، کلاس "bank teller" نیاز به برقراری ارتباط با کلاس‌های دیگر همانند کلاس «مشتری»، کلاس «پرداخت نقدی»، کلاس «پس‌انداز» و غیره دارد. به این روابط، *وابستگی* گفته می‌شود.

بسته کردن (package) نرم‌افزار بصورت کلاس‌ها می‌تواند ویژگی استفاده مجدد از نرم‌افزار را عرضه کند. غالباً کلاس‌های مرتبط با یکدیگر را بصورت *کامپونت‌های* با قابلیت استفاده مجدد بسته‌بندی می‌کنند.

مهندسی نرم‌افزار

استفاده مجدد از کلاس‌های موجود به هنگام ایجاد کلاس‌ها و برنامه‌های جدید سبب صرفه‌جویی در زمان و هزینه‌ها می‌شود. همچنین به برنامه‌نویسان در ایجاد سیستم‌های قابل اعتمادتر و کاراتر کمک می‌کند، چرا که کلاس‌ها و کامپونت‌های موجود غالباً از مرحله تست، خطایابی و میزان کارایی عبور کرده‌اند.

مقدمه‌ای بر تحلیل و طراحی شی‌گرا (OOAD)

بزودی شروع به برنامه‌نویسی در C++ خواهید کرد. چگونه می‌خواهید کد برنامه‌های خود را ایجاد کنید؟ شاید، می‌خواهید همانند هر برنامه‌نویس تازه‌کاری، فقط کامپیوتر را روشن کرده و شروع به تایپ برنامه کنید. این روش می‌تواند در ارتباط با برنامه‌های کوچک کاربرد داشته باشد، اما اگر از شما خواسته شود تا یک سیستم نرم‌افزاری برای کنترل صدها دستگاه ماشین پرداخت اتوماتیک یک بانک ایجاد کنید، چه خواهید کرد؟ یا اگر از شما خواسته شود تا با یک تیم هزار نفری در ارتباط با تولید نسل بعدی سیستم کنترل ترافیک هوایی ایالات متحده کار کنید، چه خواهید کرد؟ برای پروژه‌های به این بزرگی و پیچیدگی، نمی‌توانید به راحتی پشت کامپیوتر نشسته و شروع به برنامه‌نویسی کنید.

برای دستیابی به بهترین راه حل، بایستی بدقت نیازمندیهای پروژه را تجزیه و تحلیل کنید (تعیین کنید که سیستم چه کار می‌خواهد انجام دهد) و طرحی را توسعه دهید که آنها را برآورده سازد (سیستم قادر به تصمیم‌گیری صحیح برای انجام وظایف خود باشد). در حالت ایده‌آل، قبل از هرگونه کدنویسی، باید به سراغ فرآیند رفته و به دقت طراحی را انجام دهید. اگر این فرآیند مستلزم تحلیل و طراحی سیستم از نقطه نظر شی‌گرا باشد، آن را ***object-orientend analysis and design (OOAD)*** یا ***تحلیل و طراحی شی‌گرا*** می‌گویند. برنامه‌نویسان با تجربه مطلع هستند که تحلیل و طراحی می‌تواند در زمان و هزینه ایجاد برنامه بسیار صرفه‌جویی کند، با اجتناب از اعمال طرح‌های ضعیف که در هر بار برنامه را به شکست کشانده و باعث می‌شوند کار از ابتدا آغاز گردد که همان تحمیل هزینه و زمان است.

OOAD یک کلمه کلی برای بیان فرآیند تجزیه و تحلیل یک مسئله و توسعه روش حل آن مسئله است. مسائل کوچکی همانند مسائلی که در چند فصل اولیه با آنها برخورد خواهیم داشت، نیازی به فرآیند OOAD ندارند. برای ایجاد این برنامه‌ها، می‌توان ابتدا مبادرت به نوشتن *شبه کد* آنها کرد و سپس کد C++ آنها را نوشت. در فصل ۴ به معرفی شبه کد خواهیم پرداخت.

با بزرگتر و پیچیده‌تر شدن مسئله، برتری روش OOAD به روش شبه کد کاملاً مشخص می‌شود. اگر چه پردازش‌های مختلفی از OOAD وجود دارد، اما یک زبان گرافیکی برای نمایش نتایج هر فرآیند OOAD بیشتر از همه بکار گرفته شده است. این زبان *UML (Unified Modeling Language)* نام دارد که اواسط دهه ۱۹۹۰ تحت نظر سه نظریه‌پرداز نرم‌افزار بنام‌های Grady Booch، James Rumbaugh و Ivar Jacobson توسعه پیدا کرده است.

تاریخچه UML

در دهه ۱۹۸۰، تعداد سازمان‌هایی که شروع به استفاده از OOP برای ایجاد برنامه‌های کاربردی خود کردند بسیار افزایش یافت و از اینرو نیاز به یک استاندارد OOAD احساس گردید. تعدادی از نظریه‌پردازان شامل Booch، Rumbaugh و Jacobson، بطور جداگانه مبادرت به ایجاد و عرضه فرآیندهای برای برطرف کردن این نیاز کردند. هر فرآیند یا پردازه دارای نمادها یا «زبان» (به فرم دیاگرام‌های گرافیکی) متعلق بخود بود، تا حاصل نتایج تجزیه و تحلیل و طراحی‌ها باشد.

در اوایل دهه ۱۹۹۰، سازمان‌های مختلف و حتی قسمتهای موجود در درون یک سازمان، در حال استفاده از فرآیندهای منحصر بفرد با نمادهای متعلق بخود بودند. همزمان، این سازمان‌ها مایل به استفاده از ابزارهای نرم‌افزاری بودند که از فرآیندهای خاص آنها پشتیبانی کند. سازندگان نرم‌افزار متوجه مشکل تهیه ابزارهایی برای کار با این همه فرآیند متفاوت شدند. واضح بود که نیاز به یک نماد و فرآیند استاندارد بوجود آمده است.

در سال ۱۹۹۴، James Rumbaugh با همکاری Grady Booch در شرکت Rational Softwave (که اکنون بخشی از IBM است)، شروع به متحدالشکل کردن فرآیندهای خود کرد. بلافاصله Ivar Jacobson نیز به آنها پیوست. در سال ۱۹۹۶، گروه مبادرت به عرضه نسخه اولیه از UML به جامعه مهندسین نرم‌افزار کرد و پاسخ آنها را خواستار شدند. در همان زمان، سازمانی که بعنوان *OMG (Object Management Group)* شناخته می‌شود، دعوت به یک زبان مدل‌سازی مشترک از گروه‌های مختلف کرد. OMG (به آدرس www.omg.org) یک سازمان غیرتجاری است که مبادرت به نشر تکنولوژی‌های استاندارد شی‌گرا همانند UML می‌کند. چندین شرکت، از جمله HP، IBM، Microsoft، Oracle و Rational Software تشخیص داده بودند که نیاز به یک زبان مدل‌سازی مشترک است. در پاسخ به تقاضای OMG، این شرکت‌ها، همکاری UML خود را آغاز کردند، کنسرسیوم UML نسخه 1.1 را عرضه و تحویل OMG داد. OMG آن

را پذیرفته و در1997، مسئولیت ادامه و نگهداری UML را قبول کرد. در مارس ۲۰۰۳، OMG مبادرت به عرضه UML نسخه 1.5 کرد. UML نسخه 2 که در زمان نشر این کتاب مراحل پایانی خود را می‌گذارند، اصلی‌ترین نسخه از سال ۱۹۹۷ است. از اینرو بحث ما بر روی UML نسخه 2 خواهد بود.

UML چیست؟

در حال حاضر زبان UML یکی از پرکاربردترین طرح‌های نمایش گرافیکی برای مدل کردن سیستم‌های شی‌گرا است. این زبان انواع فرآیندهای موجود را متحد کرده است. برای مدل کردن سیستم‌ها از این زبان در سراسر کتاب استفاده کرده‌ایم.

یکی از ویژگی‌های جذاب UML در انعطاف‌پذیری آن است. UML بسط‌پذیر بوده (ویژگی‌های جدید را می‌پذیرد) و از هر فرآیند یا طریقه خاص OOAD مستقل است. مدل‌کننده‌های UML برای استفاده در طراحی سیستم‌ها مجانی هستند.

UML یک زبان مرکب با ویژگی‌های گرافیکی مناسب است. در بخش‌های «مبحث آموزشی مهندسی نرم‌افزار» در توسعه نرم‌افزار ماشین پرداخت اتوماتیک (ATM) نمایش ساده و زیرمجموعه‌ای از ویژگیت‌های UML عرضه خواهیم کرد. این مباحث آموزشی به دقت و تحت نظر اساتید دانشگاه و افراد حرفه‌ای طراحی شده‌اند. امیدواریم که از مطالعه و کار کردن با این مباحث لذت ببرید.

منابع اینترنت و وب UML

- **www.uml.org**

این صفحه UML از گروه OMG تدارک بیننده مستنداتی در ارتباط با UML و تکنولوژی‌های دیگر شی‌گرا است.

- **www.ibm.com/software/rational/uml**

این صفحه UML متعلق به IBM Rational است.

کتاب‌های توصیه شده

کتاب‌های متعددی در ارتباط UML به چاپ رسیده است. کتاب‌های توصیه شده در این بخش حاوی اطلاعاتی در مورد طراحی شی‌گرا با UML هستند.

- **Arlow,J.,and I.Neustadt.*UML and the Unified Process:Practical Object-Oriented Analysis and Design*. Lodon:Pearson Education Ltd.,2002.**
- **Fowler,M.*UML Distilled, Third Edition: A Breif Guide to the Standard Object Modeling Language*. Boston:Addison-Wesley,2004.**
- **Rumbaugh,J.,I.Jacobson and G.Booch. *The Unified Modeling Language User Guide*. Reading,MA:Addison-Wesley,1999.**

برای یافتن کتاب‌های دیگری در زمینه UML، می‌توانید به وب‌سایت‌های **www.amazon.com** یا **www.bn.com** مراجعه کنید. همچنین در وب سایت IBM لیستی از کتاب‌های UML وجود دارد:

www.ibm.com/software/rational/info/technical/books.jsp

خودآزمایی بخش ۱۷-۱

۱-۱ سه نمونه از شی‌های دنیای واقعی که در مورد آنها صحبت نکردیم، بیان کنید. برای هر شی، لیستی از صفات و رفتار آن تهیه نمائید.

۱-۲ شبه کد _____

a) کلمه دیگر OOAD است.

b) یک زبان برنامه‌نویسی بکار رفته برای نمایش دیاگرام‌های UML است.

c) یک مفهوم غیررسمی از بیان منطقی برنامه.

d) نمایش گرافیکی برای مدل‌سازی سیستم‌های شی‌گرا.

۱-۳ کاربرد اصلی UML در _____

a) تست سیستم‌های شی‌گرا.

b) طراحی سیستم‌های شی‌گرا.

c) پیاده‌سازی سیستم‌های شی‌گرا.

d) a و b.

پاسخ خودآزمایی بخش ۱۷-۱

۱-۱ [نکته: پاسخ‌ها می‌توانند متفاوت باشند.]. a) صفات یک تلویزیون عبارتند از سایز صفحه نمایش، تعداد رنگ‌هایی که می‌تواند بنمایش درآورد. کانال جاری و صدای آن. تلویزیون می‌تواند روشن یا خاموش شود، کانال‌های آن عوض شود، صدا و تصویر به نمایش در آورد. b) صفات قهوه‌ساز عبارتند از حداکثر آبی که می‌تواند در خود نگهداری کند، زمان تهیه قهوه و درجه حرارت صفحه قهوه‌ساز. قهوه‌ساز می‌تواند روشن و خاموش شود، قهوه را بجوشاند. c) صفات یک لاک‌پشت عبارتند از سن، سایز پوسته (لاک) و وزن آن. یک لاک‌پشت راه می‌رود و در لاک خود پنهان می‌شود.

۱-۲ c.

۱-۳ b.

۱-۱۸ منابع وب

وب سایت Deitel & Associates

- www.deitel.com/books/cppHTP5/index.html
- www.deitel.com
- www.deitel.com/newsletter/subscribe.html
- www.prenhall.com/deitel

کامپایلرها و ابزارهای توسعه

- www.thefreecountry.com/developercity/ccompilers.shtml
- msdn.microsoft.com/visualC
- www.borland.com/bcppbuilder
- www.compilers.net

- www.kai.com/C_plus_plus
- www.symbian.com/developer/development/cppdev.html

منابع

- www.hal9k.com/cug
- www.devx.com
- www.acm.org/crossroads/xrds3-2/ovp32.html
- www.accu.informika.ru/resources/public/terse/cpp.htm
- www.cuj.com
- www.research.att.com/~bs/homepage.html

بازی‌ها

- www.codearchive.com/list.php?go=0708
- www.mathools.net/c_c__/Games/
- www.gametutorials.com/Tutorials/GT/GT_Pg1.htm
- www.forum.nokia.com/main/0,6566,050_20,00.htm

خودآزمایی

۱-۱ جاهای خالی در عبارات زیر را با کلمات مناسب پر کنید:

a) شرکت ..... سبب محبوبیت کامپیوترهای شخصی شد.

b) کامپیوتر ..... به نحوی ساخته شده بود که از قابلیت محاسبات شخصی و کار در صنایع برخوردار بود.

c) کامپیوترها پردازش داده‌ها را تحت کنترل مجموعه‌ای از دستورالعمل‌ها بنام ..... کامپیوتری انجام می‌دهند.

d) شش واحد منطقی و کلیدی در یک کامپیوتر عبارتند از ..... ، ..... ، ..... ، ..... ، ..... ، و .....

e) سه کلاس زبان معرفی شده در این فصل عبارت بودند از ..... ، ..... و .....

f) برنامه‌های که مبادرت به ترجمه برنامه‌های زبان سطح بالا به زبان ماشین می‌کنند، ..... نام دارند.

g) از زبان C بطرز گسترده‌ای در توسعه سیستم عامل ..... استفاده شده است.

h) ..... زبانی است که توسط Wirth توسعه پیدا کرد و هدف از آن آموزش برنامه‌نویسی در دانشگاه‌ها است.

i) دپارتمان دفاع توسعه دهنده زبان Ada با قابلیتی بنام ..... است که به برنامه‌نویسان امکان می‌دهد تا تعدادی از فعالیت‌ها را بصورت موازی به انجام برسانند.

۱-۲ جاهای خالی در عبارات زیر که در ارتباط با محیط C++ هستند با کلمات مناسب پر کنید:

a) معمولا برنامه‌های C++ توسط یک برنامه ..... تایپ شده و وارد کامپیوتر می‌شوند.

b) در یک سیستم C++، برنامه ..... قبل شروع بکار فاز ترجمه کامپایلر اجرا می‌شود.

c) برنامه ..... مبادرت به ترکیب خروجی کامپایلر با انواع مختلفی از توابع کتابخانه‌ای برای تولید یک حالت اجرائی می‌کند.

d) برنامه ..... حالت اجرائی یک برنامه C++ را از روی دیسک به حافظه منتقل می‌کند.

۱-۳ جاهای خالی را که در ارتباط با مبحث شی‌گرا هستند با کلمات مناسب پر کنید:

a) شی‌ها دارای ویژگی ..... هستند، با اینکه شی‌ها از نحوه برقراری ارتباط با یک شی دیگر از طریق واسط مطلع هستند، معمولا از نحوه ساختار داخلی سایر شی‌ها اطلاعی ندارند.

b) برنامه‌نویسان C++ متمرکز بر ایجاد ..... هستند که حاوی اعضای داده و توابع عضو می‌باشند.

c) کلاس‌ها می‌تواند با سایر کلاس‌ها رابطه داشته باشند. این رابطه ..... نامیده می‌شود.

d) از منظر شی‌گرا، فرآیند تحلیل و طراحی یک سیستم ..... نامیده می‌شود.

e) OOD از مزیت رابطه ..... سود می‌برد، که در آن شی‌های جدید صفات کلاس‌های قبلی را جذب می‌کنند.

f) زبان ..... یک زبان گرافیکی است که طراحان سیستم‌های نرم‌افزاری امکان نمایش استاندارد طرح‌ها را فراهم می‌آورد.

g) سایز، شکل، رنگ و وزن یک شی نشاندهنده ..... آن شی است.

پاسخ خودآزمایی

۱-۱ a) اپل، b) کامپیوتر شخصی IBM ، c) برنامه، d) واحد ورودی، واحد خروجی، واحد حافظه، واحد محاسبه و منطق، واحد پردازش مرکزی، واحد ذخیره‌سازی ثانویه. e) زبان ماشین، زبان اسمبلی، زبان سطح بالا. f) کامپایلرها. g) یونیکس. h) پاسکال. i) multitasking

۱-۲ a) ویرایشگر. b) پیش‌پردازنده. c) linker d) loader

۱-۳ a) پنهان‌سازی اطلاعات b) کلاس‌ها c) وابستگی d) تحلیل و طراحی شی‌گرا (OOD). e) توارث f) UML g) صفت

تمرینات

۱-٤ هر کدامیک از ایتم‌های زیر را در دو دسته نرم‌افزار و سخت‌افزار رده‌بندی کنید:

a) CPU

b) کامپایلر C++

c) ALU

d) پیش‌پردازنده C++

e) واحد ورودی

f) برنامه ویرایشگر

۱-۵ به کدام دلیل مایل هستید تا برنامه‌ای بنویسید که مستقل از زبان ماشین باشد بجای اینکه وابسته به زبان ماشین باشد؟ چرا یک زبان وابسته به ماشین در نوشتن برخی از برنامه‌های خاص مناسب است؟

۱-٦ جاهای خالی را در عبارات زیر با کلمات مناسب پر کنید:

a) کدام واحد منطقی کامپیوتر اطلاعات خارج از آن را برای استفاده کامپیوتر دریافت می‌کند؟.....

b) پردازش دستورالعمل توسط کامپیوتر برای حل مسئله‌ای ..... نامیده می‌شود.

c) کدام نوع از زبان کامپیوتر از زبانی شبیه به زبان مخفف شده انگلیسی در دستورالعمل‌های زبان ماشین استفاده می‌کند؟.....

d) کدام واحد منطقی کامپیوتر اطلاعات پردازش شده توسط خود را به به دستگاه‌های دیگر که احتمالا در خارج از کامپیوتر هم می‌توانند قرار داشته باشند، ارسال می‌کند؟.....

e) کدام واحد منطقی کامپیوتر مبادرت به نگهداری اطلاعات می‌کند؟.....

f) کدام واحد منطقی کامپیوتر مسئول انجام محاسبات است؟.....

g) کدام واحد منطقی کامپیوتر مسئول تصمیم‌گیری است؟.....

h) برنامه‌نویسی با سطح زبان ..... برای بسیاری از برنامه‌نویسان مناسب بوده و تولید برنامه در آنها سریعتر صورت می‌گیرد.

i) زبان ..... تنها زبانی است که یک کامپیوتر می‌تواند مستقیما آن را درک کند.

j) کدام واحد منطقی کامپیوتر مسئول هماهنگی مابین سایر قسمتهای منطقی است؟.....

۱-۷ به چه دلایلی این روزها توجه زیادی به برنامه‌نویسی شی‌گرا و انجام آن بویژه توسط C++ می‌شود؟

۱-۸ وجه تمایز مابین عبارت خطای عظیم با خطای غیر عظیم در چیست؟

فصل دوم

مقدمه‌ای بر برنامه‌نویسی C++

اهداف

- **نوشتن برنامه‌های ساده کامپیوتری در C++.**
- **نوشتن عبارات ساده ورودی و خروجی.**
- **استفاده از نوع‌های بنیادین.**
- **مفاهیم حافظه.**
- **استفاده از عملگرهای محاسباتی.**
- **نوشتن عبارات ساده تصمیم‌گیری.**

رئوس مطالب

۲-۱ مقدمه

در این فصل به معرفی برنامه‌نویسی C++ می‌پردازیم، که طراحی برنامه‌ها را تسهیل خواهد بخشید. اکثر برنامه‌های که در این کتاب با آنها مواجه خواهید شد، مبادرت به پردازش اطلاعات و نمایش نتایج می‌کنند. در این فصل، به معرفی پنج مثال می‌پردازیم که نحوه نمایش پیغام و همچنین دریافت داده از کاربران را به شما نشان می‌دهند. سه مثال اول، ساده بوده و تنها مبادرت به نمایش پیغام در صفحه نمایش می‌کنند. برنامه بعدی، دو عدد از کاربر دریافت کرده و مجموع آنها را محاسبه و نتیجه را بنمایش درمی‌آورد. در ضمیمه بحث، شما را با نحوه انجام انواع محاسبات و ذخیره نتایج برای استفاده‌های بعدی آشنا خواهیم کرد. مثال پنجم در ارتباط با تصمیم‌سازی است که مبادرت به مقایسه دو عدد کرده و سپس پیغامی براساس مقایسه بنمایش درمی‌آورد. برای اینکه درک مناسب و آسانی از برنامه‌نویسی C++ بدست آورید، خط به خط هر برنامه را تحلیل می‌کنیم. برای اینکه مهارت‌های کسب کرده خود از این فصل را بکار گیرید، چند مسئله برنامه‌نویسی در بخش تمرینات در نظر گرفته‌ایم.

۲-۲ یک برنامه ساده: چاپ یک عبارت متنی

زبان C++ از نمادهای استفاده می‌کند که ممکن است برای غیر برنامه‌نویسان عجیب و غریب بنظر برسند. در این بخش به بررسی برنامه‌ای می‌پردازیم که عبارتی را در یک خط چاپ می‌کند. برنامه و خروجی آن در شکل ۲-۱ آورده شده است. این برنامه حاوی چندین ویژگی مهم زبان C++ است. برای آشنایی بهتر شما، به توضیح خط به خط برنامه می‌پردازیم.

در خطوط 1 و 2

```
// Fig. 2.1: fig02_01.cpp
// Text-printing program.
```

که با //، آغاز شده‌اند، نشان‌دهنده این مطلب هستند که این خطوط *توضیح* می‌باشند. برنامه‌نویسان توضیحات را در برنامه یا لیست کد قرار می‌دهند تا خوانایی کدهای خود را افزایش دهند. توضیحات می‌توانند در خط متعلق به خود جای داده شوند که آنها را *توضیحات تمام خط* می‌نامیم یا در انتهای یک خط از کد که آنها را توضیحات *انتهای خط می‌نامیم*، قرار بگیرند. کامپایلر C++ توضیحات را نادیده می‌گیرد، به این معنی که توضیحات هیچ تأثیری در اجرای برنامه ندارند. توضیح بکار رفته در خط 1 فقط نشان‌دهنده شماره تصویر و نام فایل این برنامه است. البته برخی از برنامه‌نویسان عبارات توضیحی خود را مابین کاراکترهای */ و /* قرار می‌دهند.

```
1  // Fig. 2.1: fig02_01.cpp
2  // Text-printing program.
3  #include <iostream> // allows program to output data to the screen
4
5  // function main begins program execution
6  int main()
7  {
8     std::cout << "Welcome to C++!\n"; // display message
9
10    return 0; // indicate that program ended successfully
11
12 } // end function main
```

```
Welcome to C++!
```

شکل ۱-۲ | برنامه چاپ عبارت.

برنامه‌نویسی ایده‌ال

هر برنامه‌ای باید با یک یا چندین توضیح در ارتباط با اهداف برنامه و همچنین نام برنامه‌نویس، تاریخ و زمان آغاز شود.

خط 3

```
#include <iostream> // allows program to output data to the screen
```

یک رهنمود *پیش‌پردازنده* است، که پیغامی برای پیش‌پردازنده C++ می‌باشد. خطوطی که با نماد # آغاز می‌شوند، قبل از اینکه برنامه کامپایل شود توسط پیش‌پردازنده پردازش می‌شوند. این خط به پیش‌پردازنده اعلان می‌کند تا *محتویات سرآیند جریان ورودی/خروجی فایل* <***iostream***> را در برنامه وارد سازد. بایستی این فایل در هر برنامه‌ای که می‌خواهد داده‌ای به صفحه نمایش انتقال دهد، یا اینکه از صفحه‌کلید و با استفاده از جریان ورودی/خروجی C++ داده‌ای دریافت نماید، وارد شود. در شکل۱-۲ خروجی برنامه آورده شده است. در فصل ششم با فایل‌های سرآیند در فصل پانزدهم با محتویات **iostream** آشنا خواهید شد.

خطای برنامه‌نویسی

در صورتیکه افزودن فایل سرآیند <iostream> به برنامه‌ای که مبادرت به دریافت داده از صفحه کلید یا ارسال داده به صفحه نمایش می‌کند، فراموش شود، کامپایلر یک پیغام خطا ارسال خواهد کرد.

خط 4 فقط یک خط خالی است. برنامه‌نویسان برای اینکه قرائت‌های خود را آسانتر کنند از خطوط خالی، کاراکترهای فاصله (space) و تب (tab) استفاده می‌کنند. به مجموعه این کاراکترها، white space می‌گویند. معمولاً این کارکترها توسط کامپایلر نادیده گرفته می‌شوند. در این فصل و چند فصل بعدی، در ارتباط با قواعد رایج در استفاده از کاراکترهای white-space صحبت خواهیم کرد تا خوانایی برنامه‌ها افزایش یابد.

برنامه‌نویسی ایده‌ال

با استفاده از خطوط خالی و کاراکترهای فاصله، خوانایی برنامه‌ها را افزایش دهید.

خط 5

```
// function main begins program execution
```

هم یک خط توضیحی تمام خط است که اعلان می‌کند برنامه از خط بعدی شروع می‌شود.

خط 6

```
int main()
```

بخشی از هر برنامه C++ است. پرانتزهای واقع پس از **main** نشان می‌دهند که **main** یک بلوک برنامه بنام *تابع* است. برنامه‌های C++ می‌توانند حاوی یک یا چندین تابع و کلاس باشند، اما بایستی یکی از آنها حتما **main** باشد، حتی اگر **main** اولین تابع در برنامه نباشد. کلمه کلیدی **int** که در سمت چپ **main** قرار گرفته، بر این نکته دلالت دارد که **main** یک مقدار صحیح "برمی‌گرداند" (عدد بدون اعشار). کلمه‌کلیدی، کلمه‌ای در کد است که توسط C++ رزرو شده است. لیست کامل کلمات کلیدی C++ در جدول شکل ۳-۴ آورده شده‌اند. به هنگام مطالعه فصل سوم و ششم به توضیح مفهوم دقیق "مقدار برگشتی" از سوی یک تابع خواهیم پرداخت. اما برای این لحظه، کافیست بدانید که کلمه کلیدی int در سمت چپ برنامه‌های شما قرار خواهد گرفت.

بایستی براکت چپ، {، (خط 7) در ابتدای بدنه هر روالی قرار داده شود. براکت متناظر، براکت راست، }، (خط 12) است،که باید آنرا در انتهای بدنه هر روالی قرار داد. خط 8

```
std::cout << "Welcome to C++!\n"; // display message
```

به کامپیوتر *فرمان* می‌دهد تا رشته‌ای از کاراکترها را که مابین جفت گوتیشن قرار دارند بر روی صفحه نمایش چاپ کند. گاهی اوقات رشته بنام رشته کاراکتری، پیغام یا رشته *لیترال* هم خوانده می‌شود. کاراکترهای *white space* توسط کامپایلر نادیده گرفته می‌شوند.

کل خط 8، شامل **std::cout**، عملگر <<، رشته "**Welcome to C++ ! \n**" و یک سیمیکولن (;) است، که به کل این خط یک *عبارت* گفته می‌شود. هر عبارتی در C++ باید با یک سیمیکولن خاتمه پذیرد (که به آن *خاتمه‌دهنده عبارت* هم گفته می‌شود). رهنمودهای پیش‌پردازنده (همانند **#include**) با سیمکولن خاتمه نمی‌یابند. خروجی و ورودی در C++ با *جریانی* (*stream*) از کاراکترها پیاده‌سازی می‌شود. بنابر این، زمانیکه عبارت قبلی اجرا می‌شود، مبادرت به ارسال جریانی از کاراکترهای ***Welcome to C++ !*** به *شی جریان خروجی استاندارد* (**std::cout**) که معمولا با صفحه نمایش مرتبط است می‌کند. در فصل پانزدهم به بررسی بیشتر **std::cout** خواهیم پرداخت.

دقت کنید که **std::** قبل از **cout** قرار گرفته است. انجام این عمل به هنگام استفاده از رهنمود پیش‌پردازنده **#include <iostream>** الزامی است. نماد **std::cout** نشان می‌دهد که در حال استفاده از یک نام هستیم، که این نام در این مورد cout می‌باشد، که متعلق به "*فضای‌نامی*" **std** است. فضاهای‌نامی از ویژگیهای پیشرفته C++ هستند. در فصل بیست‌وچهارم بطور کامل با فضاهای‌نامی آشنا خواهید شد. اما برای این لحظه، باید بخاطر داشته باشید که کلمه **std::** را قبل از هر کدامیک از نمادهای **cout**، **cin** و **cerr** در یک برنامه قرار دهید. در برنامه شکل ۱۳-۲ از این نمادها به همراه عبارت using استفاده شده است، که ما را قادر به حذف **std::** قبل از هر استفاده از فضای نامی **std** می‌کند.

عملگر << نشاندهنده، *عملگر درج جریان* است. هنگامی که این برنامه اجرا می‌شود، مقدار موجود در سمت راست این عملگر، *عملوند* سمت راست، وارد جریان خروجی می‌گردد. دقت کنید که عملگر مستقیماً به مکانی اشاره می‌کند که داده باید به آنجا برود. معمولا کاراکترهای قرار گرفته در سمت راست عملگر به همان شکلی که مابین جفت گوتیش‌ها آورده شده‌اند، چاپ می‌شوند. با این وجود، توجه نمائید که کاراکتر **n**\ بر روی صفحه نمایش چاپ نمی‌شود. کاراکتر \ کاراکتر *escape* نامیده می‌شود. این کاراکتر نشان می‌دهد که یک کاراکتر *ویژه* چاپ خواهد شد. زمانیکه در دنباله‌ای از رشته‌های کاراکتری یک کاراکتر \ وارد شود، کاراکتر پس از آن بعنوان یک *توالی escape* در نظر گرفته خواهد شد. در این برنامه توالی escape کاراکتر **n**\ است، که به مفهوم *خط جدید* یا *newline* می‌باشد. این کاراکتر سبب

می‌شود تا کرسر به ابتدای خط بعدی در صفحه نمایش منتقل شود. در جدول شکل ۲-۲ تعدادی از توالی‌های escape که از کاربرد بیشتری برخوردار هستند آورده شده‌اند.

توالیescape	توضیحات
\n	خط جدید. کرسر را در ابتدای خط بعدی قرار می‌دهد.
\t	تب افقی. کرسر را به اندازه یک تب(tab) به جلو انتقال می‌دهد.
\r	*enter*. کرسر را به ابتدای خط جاری باز می‌گرداند. آنرا با \n اشتباه نگیرید.
\a	هشدار. زنگ یا صدای سیستم را فعال می‌کند.
\\	کاراکتر \. برای چاپ کاراکتر \ استفاده می‌شود.
\'	کاراکتر '. برای چاپ کارکتر ' استفاده می‌شود.
\"	جفت گوتیشن. از این توالی برای چاپ کاراکتر جفت گوتیشن استفاده می‌شود.

شکل ۲-۲ |توالی escape.

خطای برنامه‌نویسی

فراموش کردن سیمیکولن در پایان یک عبارت، خطای نحوی (syntax error) بدنبال خواهد داشت. این نوع خطا زمانی رخ می‌دهد که کامپایلر با عبارتی مواجه شود و نتواند مفهوم آنرا تشخیص دهد. معمولا در چنین حالتی کامپایلر یک پیغام خطا ارسال می‌کند تا به برنامه‌نویس در یافتن مکان خطا و رفع آن کمک نماید. خطاهای نحوی به هنگام خروج از قواعد نگارش زبان رخ می‌دهند. گاهاً به این خطاها، خطاهای کامپایل هم گفته می‌شود، چراکه این خطاها در زمان فاز کامپایل شدن برنامه رخ می‌دهند.

خط 10

```
return 0; // indicate that program ended successfully
```

در پایان هر تابع **main** وجود خواهد داشت. کلمه کلیدی **return** یکی از چندین روش موجود برای *خروج از یک تابع* است. زمانیکه عبارت **return** در پایان تابع **main** بکار گرفته می‌شود، همانند این برنامه، مقدار **0** نشاندهنده این مطلب است که برنامه با موفقیت به کار خود پایان داده است. در فصل ششم، با جزئیات عملکرد توابع و دلایل افزودن چنین عباراتی به آنها بیشتر آشنا خواهید شد. اما برای این لحظه، کافیست بدانید که این عبارت در انتهای هر برنامه‌ای قرار داده می‌شود. براکت راست، **}**، (خط 12) انتهای تابع **main** را نشان می‌دهد.

برنامه‌نویسی ایده‌ال

بسیاری از برنامه‌نویسان در انتهای آخرین کاراکتر چاپ شده توسط یک تابع، کاراکتر \n را قرار می‌دهند. با این عمل مطمئن می‌شوند که تابع در پایان کار خود، کرسر را در ابتدای خط بعدی قرار خواهد داد. این عمل می‌تواند در راستای ایجاد برنامه‌هایی با قابلیت استفاده مجدد موثر باشد.

برنامه‌نویسی ایده‌ال

به دندانه‌دار بودن خطوط کد دقت کنید. این عمل یکی از قواعد بکارگیری فاصله‌ها در برنامه است. دندانه‌دار کردن کد برنامه‌ها، باعث افزایش خوانای و گرامر آن می‌شود.

برنامه‌نویسی ایده‌ال

برای حفظ ظاهر آراسته و مرتب در میان کدهای نوشته شده، بهتر است میزان دندانه‌گذاری مورد نظر خود را از همان ابتدا مشخص سازید. پیشنهاد می‌کنیم تا از کلید tab با فاصله گذاری ¼ اینچ یا سه فاصله (space) از سطح دندانه‌گذاری شده فوقانی استفاده کنید.

۲-۳ اصلاح برنامه

این بخش ادامه‌دهنده معرفی برنامه‌نویسی C++ با دو مثال است، که نحوه اصلاح برنامه مطرح شده در شکل ۱-۲ را نشان می‌دهند. در مثال اول، متن در یک خط با استفاده از چندین عبارت چاپ می‌شود. در مثال دوم همان متن توسط یک عبارت و در چندین خط چاپ می‌گردد.

چاپ متن در یک خط توسط چند عبارت

رشته Welcome to C++! می‌تواند به چندین روش چاپ شود. برای مثال، در برنامه شکل ۳-۲ از دو عبارت برای وارد کردن جریان (خطوط 8-9)استفاده شده است، تا همان خروجی برنامه شکل ۱-۲ تولید شود. خروجی برنامه دوم دقیقاً همانند برنامه اول است، چرا که هر عبارت از مکانی که عبارت قبلی به عمل چاپ پایان داده، شروع به چاپ می‌کند. عبارت اول کلمه Welcome و بدنبال آن یک فاصله چاپ کرده و عبارت دوم هم کار چاپ را از همان خط و پس از فاصله چاپ شده دنبال می‌کند. بطور کلی، C++ به برنامه‌نویس امکان می‌دهد تا عبارات را به روش‌های گوناگون بکار گیرد.

چاپ متن در چند خط توسط یک عبارت

همانند برنامه شکل ۴-۲ می‌توان یک عبارت را در چندین خط به نمایش در آورد. هر بار که عبارت ارسالی به خروجی با کاراکتر توالی **\n** مواجه شود، کرسر به ابتدای خط بعدی منتقل خواهد شد. برای ایجاد یک خط خالی در خروجی، دو کاراکتر خط جدید را پشت سرهم، همانند خط 8 قرار دهید.

```
1  // Fig. 2.3: fig02_03.cpp
2  // Printing a line of text with multiple statements.
3  #include <iostream> // allows program to output data to the screen
4
5  // function main begins program execution
6  int main()
7  {
8     std::cout << "Welcome ";
9     std::cout << "to C++!\n";
10
11    return 0; // indicate that program ended successfully
12
13 } // end function main
```

```
Welcome to C++!
```

شکل ۳-۲ | چاپ متن در یک خط با چند عبارت.

```
1  // Fig. 2.4: fig02_04.cpp
2  // Printing multiple lines of text with a single statement.
3  #include <iostream> // allows program to output data to the screen
4
5  // function main begins program execution
6  int main()
7  {
8     std::cout << "Welcome\nto\n\nC++!\n";
9
10    return 0; // indicate that program ended successfully
11
12 } // end function main
```

```
Welcome
to

C++!
```

شکل ٤-۲ | چاپ متن در چند خط با یک عبارت.

٤-۲ یک برنامه ساده دیگر: جمع اعداد صحیح

در برنامه بعدی از شی جریان ورودی **std::cin** و عملگر استخراج <<، برای بدست آوردن دو عدد صحیح تایپ شده از سوی کاربر از طریق صفحه‌کلید، جمع آنها و نمایش نتیجه بدست آمده با استفاده از **std::cout** می‌پردازیم. برنامه شکل ۵-۲ عمل جمع و خروجی حاصل از این برنامه را نشان می‌دهد.

```
1  // Fig. 2.5: fig02_05.cpp
2  // Addition program that displays the sum of two numbers.
3  #include <iostream> // allows program to perform input and output
4
5  // function main begins program execution
6  int main()
7  {
8     // variable declarations
9     int number1; // first integer to add
10    int number2; // second integer to add
11    int sum; // sum of number1 and number2
12
13    std::cout << "Enter first integer: "; // prompt user for data
14    std::cin >> number1; // read first integer from user into number1
15
16    std::cout << "Enter second integer: "; // prompt user for data
17    std::cin >> number2; // read second integer from user into number2
18
19    sum = number1 + number2; // add the numbers; store result in sum
20
21    std::cout << "Sum is " <<sum <<std::endl; //display sum; end line
22
23    return 0; // indicate that program ended successfully
24
25 } // end function main
```

```
Enter first integer: 45
Enter second integer: 72
Sum is 117
```

شکل ۵-۲ | برنامه جمع که مجموع دو عدد صحیح دریافتی از صفحه کلید را محاسبه می‌کند.

توضیحات موجود در خطوط 1 و 2

```
// Fig. 2.5: fig02_05.cpp
// Addition program that displays the sum of two numbers.
```

نشاندهنده نام فایل و هدف برنامه هستند. رهنمود پیش‌پردازنده C++

```
#include <iostream> // allows program to perform input and output
```

در خط 3 حاوی محتویات سرآیند فایل ***iostream*** در برنامه است.

همانطوری که قبلاً هم گفته شد، هر برنامه‌ای با اجرای تابع **main** آغاز می‌شود(خط 6). براکت سمت چپ (خط 7) ابتدا و آغاز بدنه **main** و براکت متناظر سمت راست (خط 25)، انتها و پایان بدنه را نشان می‌دهد.

خطوط 9-11

```
   int number1; // first integer to add
   int number2; // second integer to add
   int sum; // sum of number1 and number2
```

بخش *اعلان* می‌باشند. کلمات **number1**، **number2** و **sum** اسامی *متغیرها* هستند. متغیرها مکان‌های در حافظه کامپیوتر هستند که می‌توان مقادیری را برای استفاده برنامه در آنها ذخیره کرد. این اعلان‌ها مشخص می‌کنند که متغیرهای **number1**، **number2** و **sum** از نوع داده **int** هستند، به این معنی که این متغیرها قادر به نگهداری مقادیر صحیح همانند اعداد 31914 ,0 ,11- ,7 و غیره می‌باشند. تمامی متغیرها بایستی دارای نام و یک نوع داده باشند تا بتوان از آنها در برنامه استفاده کرد. چندین متغیر از یک نوع را می‌توان در خط یا چند خط مجزا از هم اعلان کرد. می‌توانیم هر سه متغیر فوق را در یک خط و بصورت زیر اعلان کنیم:

```
   int number1, number2, sum;
```

با این همه، در این حالت خوانائی برنامه کاهش یافته و مانع می‌شود تا توضیحات مناسب برای هر متغیر و هدف از آن متغیر در برنامه را وارد سازیم. اگر بخواهیم بیش از یک متغیر در یک خط اعلان کنیم

(همانند مورد بالا) باید اسامی را با یک ویرگول(,) از یکدیگر جدا کنیم. که به این حالت *لیست جدا شده با ویرگول* گفته می‌شود.

برنامه‌نویسی ایده‌ال

پس از هر ویرگول(,) یک فاصله قرار دهید تا خوانائی برنامه افزایش یابد.

برنامه‌نویسی ایده‌ال

برخی از برنامه‌نویسان ترجیح می‌دهند تا هر متغیر را در یک خط جداگانه اعلان کنند. در این حالت قراردادن یک توضیح در کنار هر اعلان به آسانی صورت می‌گیرد.

بزودی در مورد نوع داده ***double*** (خاص اعداد حقیقی، اعداد با نقطه اعشار همانند 3.4 و 11.19-) و نوع داده ***char*** (خاص داده کاراکتری، یک متغیر از نوع **char** فقط می‌تواند یک حرف کوچک، یک حرف بزرگ، یک رقم یا یک کاراکتر خاص همانند $ یا * را در خود ذخیره سازد) صحبت خواهیم کرد.

غالباً به نوع‌های همانند **double**، **int** و **char** نوع‌های بنیادین، نوع‌های اصلی یا نوع‌های *توکار* (*built-in*) می‌گویند. انواع نوع‌های بنیادین از جمله کلمات کلیدی هستند و از اینرو باید تماماً با حروف کوچک به نمایش درآیند.

نام یک متغیر (همانند **number1**) می‌توانند هر *شناسه* معتبری که یک کلمه کلیدی نیست، باشد. یک شناسه متشکل از دنباله‌ای از کاراکترها شامل حروف، ارقام و خط زیرین (_) است. یک شناسه نمی‌تواند با یک رقم آغاز شود. زبان C++ از جمله زبان‌های *حساس به موضوع* است، به این معنی که مابین حروف کوچک و بزرگ تفاوت قائل می‌شود. از اینرو شناسه‌های a1 و A1 با هم برابر نیستند.

قابلیت حمل

زبان C++ به شناسه‌ها امکان می دهد تا هر طولی داشته باشند، اما امکان دارد سیستم شما یا ساختار C++ بکار رفته محدودیت‌هائی بر روی طول شناسه‌ها اعمال کند. از اینرو برای حفظ سازگاری و قابلیت حمل، از شناسه‌هائی با طول 31 کاراکتر یا کمتر استفاده کنید.

برنامه‌نویسی ایده‌ال

انتخاب اسامی با معنی به برنامه کمک می‌کند که خود به عنوان توضیحی بر برنامه باشد (self- documenting). در چنین حالتی اگر متن برنامه در اختیار دیگران قرار داده شود، بدون اینکه نیازی به راهنما و توضیحات اضافی باشد، عملکرد برنامه مشخص خواهد بود.

برنامه‌نویسی ایده‌ال

از اختصار یا کوته‌سازی شناسه‌ها اجتناب کنید، تا خوانایی برنامه افزایش یابد.

برنامه‌نویسی ایده‌ال

از ایجاد شناسه‌های که با یک خط زیرین یا دو خط زیرین آغاز می‌شوند اجتناب کنید، چراکه امکان دارد کامپایلر C++ از اسامی مشابه‌ای برای انجام مقاصد داخلی خود استفاده کرده باشد. در اینصورت از ایجاد تداخل مابین خود و کامپایلر جلوگیری خواهید کرد.

اجتناب از خطا

زبان‌های همانند C++ همیشه در حال بسط و گسترش هستند. امکان در زمان تکامل، کلمات کلیدی جدیدی به آنها افزوده شود. از کلمات مسئولیت آوری همانند object بعنوان شناسه اجتناب کنید. حتی اگر امروز کلمه object جزء کلمات کلیدی در C++ نباشد، اما می‌تواند روزی تبدیل به کلمه کلیدی گردد و در آینده برنامه شما در زمان کامپایل با خطای جدیدی مواجه شود که قبلا چنین چیزی وجود نداشت.

تقریباً می‌توان اعلان متغیرها را در هر کجای برنامه قرار داد، اما باید قبل از اینکه توسط برنامه بکار گرفته شوند، اعلان گردند. برای مثال، در برنامه شکل ۵-۲، اعلان در خط 9

```
int number1; // first integer to add
```

می‌توانست بلافاصله قبل از خط

```
std::cin >> number1; // read first integer from user into number1
```

قرار داده شود. اعلان موجود در خط 10

```
int number2; // second integer to add
```

می‌توانست بلافاصله قبل از خط 17

```
std::cin >> number2; // read second integer from user into number2
```

و اعلان خط 11

```
int sum; // sum of number1 and number2
```

می‌توانست بلافاصله قبل از خط 19

```
sum = number1 + number2; // add the numbers; store result in sum
```

اعلان گردد.

برنامه‌نویسی ایده‌ال

بهتر است در میان خطوط توضیحات از یک خط خالی استفاده شود. در اینصورت توضیحات از کد برنامه جدا شدهٔ و خوانائی برنامه افزایش می‌یابد.

برنامه‌نویسی ایده‌ال

اگر ترجیح می‌دهید که اعلان‌ها را در ابتدای یک تابع قرار دهید، آنها را از بخش عبارات اجرائی موجود در

داخل بدنه تابع و با استفاده از یک خط خالی جدا کنید تا این دو بخش از هم متمایز شوند.

خط 13

```
std::cout << "Enter first integer: "; // prompt user for data
```

مبادرت به چاپ رشته Enter first integer (بعنوان رشته لیترال نیز شناخته می‌شود) بر روی صفحه نمایش می‌کند. به چنین پیغام‌های *prompt* گفته می‌شود، چراکه به کاربر می‌گویند که چه کاری باید انجام دهد. خط 14

```
std::cin >> number1; // read first integer from user into number1
```

با استفاده از شی ورودی **cin** (از فضای نامی **std**) و عملگر <<، مقداری از صفحه‌کلید دریافت می‌کند. با استفاده از عملگر<< به همراه **std::cin** کاراکترها از یک ورودی استاندارد که معمولا صفحه‌کلید است، دریافت می‌شوند. در واقع می‌توان گفت که **std::cin** مقداری برای **number1** دریافت می‌کند.

هنگامی که برنامه اجرا شده و به عبارت فوق می‌رسد، منتظر می‌ماند تا کاربر مقداری برای متغیر **number1** وارد سازد. کاربر با تایپ یک مقدار صحیح و سپس فشردن کلید *Enter* (گاهاً به این کلید، کلید *Return* هم می‌گویند) به برنامه پاسخ می‌دهد. با این عمل مقدار تایپ شده به کامپیوتر ارسال می‌شود. سپس کامپیوتر کاراکترهای دریافت شده را به یک عدد صحیح تبدیل و این عدد (یا مقدار) را به متغیر **number1** نسبت می‌دهد. از این نقطه به بعد هر مراجعه‌ای به **number1** در این برنامه مترادف با استفاده از این مقدار خواهد بود.

شی‌های **std::cout** و **std::cin** تعامل مابین کاربر و کامپیوتر را تسهیل می‌بخشند. بدلیل اینکه این تعامل بفرم یک گفتگو است، غالباً به اینحالت محاسبه تعاملی یا محاوره‌ای گفته می‌شود.

خط 16

```
std::cout << "Enter second integer: "; // prompt user for data
```

مبادرت به چاپ Enter second integer بر روی صفحه نمایش می‌کند. این عبارت به کاربر اعلان می کند که چه کاری انجام دهد. خط 17

```
std::cin >> number2; // read second integer from user into number2
```

مقداری برای متغیر **number2** از سوی کاربر بدست می‌آورد.

عبارت تخصیص دهنده در خط 19

```
sum = number1 + number2; // add the numbers; store result in sum
```

مجموع متغیرهای **number1** و **number2** را محاسبه و نتیجه آنرا به متغیر **sum** تخصیص می‌دهد. از عملگر تخصیص = به این منظور استفاده شده است. مفهوم عبارت فوق به این مضمون است "sum مقدار خود را از number1 + number2 بدست می‌آورد." اکثر محاسبات در بین عبارات تخصیصی صورت می‌گیرند. به عملگر = و عملگر + ، عملگرهای *باینری* گفته می‌شود چرا که هر کدامیک از آنها دارای دو عملوند هستند. در مورد عبارت فوق، عملگر + ، دارای دو عملوند **number1** و **number2** است. همچنین عملگر =، دارای دو عملوند **sum** و مقدار بدست آمده از **number1 + number2** است.

برنامه‌نویسی ایده‌ال

در هر دو طرف یک عملگر باینری چند فاصله قرار دهید. فاصله‌ها باعث متمایز شدن نقش عملگر شده و خوانائی عبارتا افزایش می‌یابد.

خط 21

```
std::cout << "Sum is " << sum << std::endl; // display sum; end line
```

مبادرت به چاپ رشته کاراکتری Sum is و بدنبال آن مقدار عددی متغیر **sum** توسط عبارت **std::endl** می‌کند. **end1** کوتاه شده "end line" بوده و متعلق به فضای‌نامی **std** می‌باشد. کنترل کننده جریان **std::endl** یک خط جدید به خروجی منتقل و سپس "*بافر خروجی را خالی می‌کند.*" به این معنی که، در برخی از سیستم‌ها خروجی در ماشین انباشته می‌شود تا زمانیکه "ارزش" نمایش در صفحه نماش را پیدا کنند، عبارت **std::endl** خروجی‌های انباشته شده را مجبور می‌کند تا در یکباره به نمایش در آیند.

دقت کند که عبارت قبلی دو مقدار از انواع مختلف را خارج می‌سازد. عملگر >> از نحوه ارسال هر نوع داده به خروجی مطلع است. به روش استفاده از چندین عملگر >> در یک عبارت، عملیات زنجیر کردن، متصل کردن یا فرآیند آبشاری گفته می‌شود. از اینرو، داشتن چندین عبارت خروجی برای خارج کردن چند داده متمایز ضروری نیست.

البته می‌توان محاسبات را در بین عبارات خروجی هم انجام داد. می‌توانیم عبارات موجود در خطوط 19 و 21 را در یک عبارت، بفرم زیر داشته باشیم

```
std::cout << "Sum is " << numbeer1 + number2 << std::endl;
```

بنابر این دیگر نیازی به متغیر **sum** نخواهد بود. براکت راست، **}**، به کامپیوتر اطلاع می‌دهد که به انتهای تابع **main** رسیده است.

از جمله توانمندیهای C++ این است که به کاربران اجازه می‌دهد تا نوع داده‌های مورد نیاز و متعلق به خود را ایجاد کنند (در فصل سوم در این مورد و در فصل‌های نهم و دهم نگاهی دقیق به این موضوع خواهیم داشت). سپس با نحوه رسیدگی C++ به هنگام کار با نوع داده‌های جدید با استفاده از عملگرهای << و >> بیشتر آشنا خواهید شد (مبحث عملگرهای سربارگذاری در فصل یازدهم).

۲-۵ مفاهیم حافظه

اسامی متغیرها، همانند **number1**، **number2** و **sum** مطابق با مکان‌های واقعی در حافظه کامپیوتر هستند. هر متغیر دارای یک نام، نوع، سایز و مقدار است. در برنامه جمع ۲-۵ زمانیکه عبارت

```
std::cin >> number1; // read first integer from user into number1
```

در خط 14 اجرا می‌شود، کاراکترهای تایپ شده توسط کاربر به یک عدد صحیح تبدیل می‌شود و در مکانی از حافظه با نام **number1** و با کمک کامپایلر C++ قرار داده می‌شود. فرض کنید که کاربر عدد 45 را به عنوان مقداری برای **number1** وارد کند. کامپیوتر 45 را دریافت و در مکان **number1** همانند شکل ۲-۶ قرار می‌دهد.

زمانیکه مقداری در یک مکان حافظه قرار می‌گیرد، مقدار جدید بر روی مقدار قبلی بازنویسی می‌شود. مجدداً به سراغ برنامه جمع می‌رویم، هنگامی که عبارت

```
std::cin >> number2; // read second integer from user into number2
```

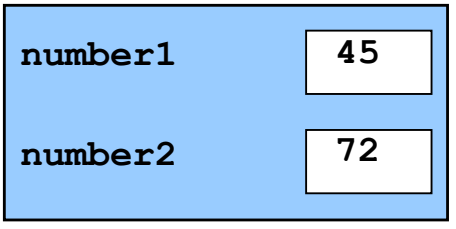
در خط 17 اجرا شود، فرض کنید کاربر مقدار 72 را وارد سازد، این مقدار وارد مکان **number2**، شده و تصویر حافظه همانند شکل ۲-۷ خواهد شد. دقت کنید که این مکان‌های حافظه ضرورتا، نبایستی در کنار هم قرار داشته باشند.

پس از اینکه برنامه مقادیر **number1** و **number2** را بدست آورد، این مقادیر را با هم جمع کرده و مجموع را در درون متغیر **sum** قرار می‌دهد. عبارت

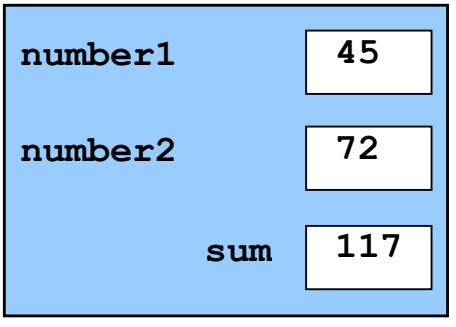
```
sum = number1 + number2; // add the numbers; store result in sum
```

علاوه بر آنکه عمل جمع را انجام می‌دهد، مقدار حاصله را هم در **sum** ذخیره می‌سازد. این عمل زمانی رخ می‌دهد که مجموع دو متغیر **number1** و **number2** محاسبه شده‌اند و در مکان **sum** ذخیره می‌شود. پس از اینکه **sum** محاسبه شد، حافظه بفرم شکل ۸-۲ تبدیل خواهد شد. توجه نمائید که مقادیر **number1** و **number2** پس از انجام محاسبه **sum** همچنان بدون تغییر باقی می‌ماند. با اینکه از این مقادیر استفاده شده است، اما مقادیر اولیه آنها با انجام عمل محاسباتی از بین نمی‌رود.

شکل ٦-٢ | مکان حافظه در حال نمایش نام و مقدار متغیر number1.



شکل ۷-۲ | مکان‌های حافظه پس از ذخیره‌سازی مقادیر برای number1 و number2.



شکل ۸-۲| مکان‌های حافظه پس از محاسبه sum با استفاده از number1 و number2.

٦-٢ محاسبات

اکثر برنامه‌ها محاسبات ریاضی انجام می‌دهند. عملگرهای ریاضی در جدول شکل ۹-۲ لیست شده‌اند. توجه کنید که تمام نمادهای بکار رفته در جبر در C++ بکار گرفته نمی‌شوند. علامت ستاره (*) نشاندهنده ضرب و علامت درصد (%) نشاندهنده باقیمانده است. اکثر عملگرهای حسابی (در جدول ۹-

۲) از نوع عملگرهای باینری هستند چرا که هر عملگر مابین دو عملوند قرار می‌گیرد. برای مثال، عبارت حسابی number1 + number2 شامل عملگر باینری + و دو عملوند **number1** و **number2** می‌باشد.

عملیات C++	عملگر محاسباتی	عبارت جبری	عبارت++C
جمع	+	$f + 7$	$f + 7$
تفریق	-	$p - c$	$p - c$
ضرب	*	bm	$b * m$
تقسیم	/	x / y یا $\frac{x}{y}$ یا $x \div y$	x / y
باقیمانده	%	$r \bmod s$	$r \% s$

شکل ۹-۲ | عملگرهای محاسباتی.

قوانین تقدم عملگر

C++ عملگرها را در عبارات محاسباتی با توالی که *قانون تقدم عملگرها* تعیین می‌کند بکار می‌برد. این قوانین شبیه قوانین موجود در جبر هستند:

۱- عملگرهایی که در درون جفت پرانتز قرار دارند دارای اولویت اول هستند. بنابر این برنامه‌نویس با استفاده از پرانتز می‌تواند ترتیب اجرای محاسبات را در دست بگیرد. پرانتزها دارای بالاترین سطح تقدم می‌باشند. در مواردی که پرانتزها به صورت تودرتو (آشیانه‌ای) قرار گرفته‌باشند، عملگر پرانتزی که در داخلی‌ترین سطح قرار دارد ابتدا انجام می‌گیرد، همانند

```
((a + b) + c)
```

عملگر قرار گرفته در جفت پرانتز داخلی ابتدا بکار گرفته می‌شود.

۲- عملگرهای ضرب و تقسیم و باقیمانده در مرحله بعدی بکار گرفته می‌شوند. اگر عبارتی شامل چندین عملگر ضرب، تقسیم و باقیمانده باشد، عملگرها از سمت چپ به راست اجرا خواهند شد. ضرب، تقسیم و باقیمانده دارای اولویت هم سطح یا برابر هستند.

۳- عملگرهای جمع و تفریق در آخرین مرحله به کار می‌روند. اگر عبارتی شامل چندین عملگر تفریق و جمع باشد، عملگرها به ترتیب از سمت چپ به راست اجرا خواهند شد. عملگرهای جمع و تفریق دارای اولویت هم سطح هستند.

وجود قوانین تقدم عملگرها، زبان C++ قادر می‌سازد تا عملگرها را با ترتیب صحیح بکار گیرد. در جدول شکل ۱۰-۲ خلاصه‌ای از قوانین تقدم عملگرها آورده شده است. این جدول با معرفی عملگرهای دیگر C++ در فصل‌های بعدی تکمیل‌تر خواهد شد. جدول کامل تقدم عملگرها در پیوست کتاب موجود است.

عملگر(ها)	عملیات	ترتیب ارزیابی (تقدم)
( )	پرانتز	اولویت اول. اگر پرانتزها به صورت تودرتو باشند، عبارتی که در درون داخلی‌ترین پرانتز قرار دارد ابتدا محاسبه می‌شود. اگر چندین جفت پرانتز در یک خط قرار گرفته باشند (تودرتو نباشند) ترتیب اجرا از سمت چپ به راست خواهد بود.
*, / , %	ضرب و تقسیم و باقیمانده	اولویت دوم. اگر چند مورد از چنین عملگرهای وجود داشته باشد، ترتیب اجرا از سمت چپ به راست خواهد بود.
+, -	جمع و تفریق	اولویت آخر. اگر چندین عملگر جمع و تفریق وجود داشته باشد ترتیب اجرا از سمت چپ به سمت راست خواهد بود.

شکل ۱۰-۲ | تقدم عملگرهای محاسباتی.

عبارات ساده جبری و C++

حال اجازه دهید تا به چند عبارت محاسباتی نگاهی بیاندازیم تا بخوبی با قوانین تقدم عملگرهای محاسباتی آشنا شوید. در هر مثالی که ذکر می‌شود عبارت جبری و معادل C++ آن عبارت نیز آورده شده است. مثال زیر یک عبارت ریاضی را نشان می‌دهد که منظور از آن به دست آوردن میانگین پنج عدد است:

$$\text{جبری: } m = \frac{a+b+c+d+e}{5}$$

```
C++ : m = (a + b + c + d + e) / 5;
```

وجود پرانتز در این عبارت ضروری است چرا که عملگر تقسیم تقدم بالاتری نسبت به عملگر جمع دارد، در نتیجه مقدار داخلی پرانتز بر 5 تقسیم می‌شود. اگر پرانتز در این عبارت حذف شود، منظور محاسبه a + b + c + d + e / 5 خواهد بود که معادل عبارت زیر است:

$$a+b+c+d+\frac{e}{5}$$

عبارت زیر نشاندهنده یک معادله است:

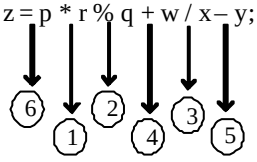
```
جبری: y = mx + b
C++ : y = m * x + b;
```

وجود پرانتز در این عبارت نیاز نیست، چرا که عملگر ضرب تقدم بالاتری نسبت به عملگر جمع دارد و در ابتدا انجام می‌شود. عمل تخصیص در آخرین مرحله صورت می گیرد چرا که به نسبت عمل ضرب و جمع از اولویت پایین‌تری برخوردار است.

مثالی که در زیر آورده شده حاوی عملگرهای توان، ضرب، تقسیم اعشاری، جمع و تفریق است:

جبری : `z = pr%q + w / x - y`

C++ : z = p * r % q + w / x – y;



دایره‌های حاوی اعداد نشاندهنده ترتیب اجرای عملگرها هستند. عملگر ضرب در اولویت اول قرار دارد و عملگرهای باقیمانده و تقسیم در اولویت‌های بعدی و به ترتیب از سمت چپ به راست اجرا می‌شوند و پس از آنها عملگرهای جمع و تفریق به ترتیب اجرا شده و در پایان عمل تخصیص صورت می‌گیرد.

توجه کنید که در این جدول در مورد پرانتزهای تودرتو مطالبی بیان شده است، اما تمام عبارات محاسباتی که دارای چندین جفت پرانتز هستند، ممکن است حاوی پرانتزهای تودرتو نباشد. برای مثال، اگر چه عبارت زیر

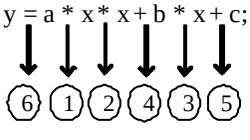
```
a * (b + c) + c * (d + e)
```

شامل دو جفت پرانتز است، اما هیچ کدامیک از آنها پرانتز تودرتو نمی‌باشند. در چنین حالتی هر دو آنها دارای سطح یکسان می‌باشند.

ارزیابی معادله درجه دوم

برای درک بهتر قوانین تقدم عملگرها به مثال زیر که یک چند جمله‌ای درجه دوم است توجه کنید:

y = a * x* x + b * x + c;



دایره‌های حاوی اعداد نشاندهنده ترتیب اجرای عملگرها هستند. در C++ عملگر محاسباتی برای انجام عمل توان وجود ندارد، از اینرو عبارت x^2 را بصورت **x * x** نشان داده‌ایم. بزودی در مورد تابع استاندارد کتابخانه‌ای ***pow*** (توان) صحبت خواهیم کرد.

خطای برنامه‌نویسی

*برخی از زبان‌های برنامه‌نویسی از عملگرهای ** یا ^ برای نمایش توان استفاده می‌کنند. در حالیکه زبان C++ از این عملگرها پشتیبانی نمی‌کند، و استفاده از آنها خطای نحوی خواهد بود.*

حال فرض کنید که a، b، c و x بصورت a = 2، b = 3، c = 7 و x = 5 مقداردهی شده باشند. با توجه به شکل ۱۱-۲ تقدم عملگرها در این چند جمله‌ای درجه دوم و نتیجه اجرای آنرا تعقیب می‌کنیم.

برنامه‌نویسی ایده‌ال

در عملیات جبری استفاده از پرانتزهای اضافی موجب می‌شود که عبارات از وضوح کافی برخوردار شوند (پرانتزهای غیرضروری redundant parentheses یا پرانتزهای افزونگی نیز نامیده می‌شوند). به کمک این پرانتزها می‌توان عبارات بزرگ و پیچیده را دسته‌بندی کرده و سبب واضح شدن طریقه انجام محاسبات شد.

۲-۷ تصمیم‌گیری: عملگرهای مقایسه‌ای و رابطه‌ای

در این قسمت به معرفی ساختار **if** در C++ می‌پردازیم که بر مبنای برقرار بودن یا نبودن برخی از *شرط‌ها* اقدام به تصمیم‌گیری می‌کند. عبارت موجود در یک ساختار **if**، شرط نامیده می‌شود. اگر شرط مورد قبول واقع شود (شرط true باشد) عباراتی که در داخل بدنه ساختار **if** قرار گرفته‌اند اجرا می‌شوند و اگر شرط مورد قبول واقع نشود عبارات داخل بدنه اجرا نخواهند شد. برای آشنائی شما با چنین ساختاری به بررسی یک مثال خواهیم پرداخت.

شرط‌های که در ساختار **if** بکار می‌روند می‌توانند از *عملگرهای مقایسه‌ای* و *رابطه‌ای* که در جدول شکل ۱۲-۲ آورده شده‌اند، استفاده کنند. تمام عملگرهای رابطه‌ای دارای اولویت یکسان بوده و از سمت چپ به راست ارزیابی می‌شوند. عملگرهای مقایسه‌ای به نسبت عملگرهای رابطه‌ای از اولویت پایین‌تری برخوردار هستند و آنها هم از سمت چپ به راست ارزیابی می‌شوند.

گام اول Y = 2 * 5 * 5 + 3 * 5 + 7;	
2 * 5= 10	**سمت چپ‌ترین ضرب**
گام دوم Y = 10 * 5 + 3 * 5 + 7;	
10 * 5 = 50	**سمت چپ‌ترین ضرب**
گام سوم Y = 50 + 3 * 5 + 7;	
3 * 5 = 15	**ضرب قبل از جمع**
گام چهارم Y = 50 + 15 + 7;	
50 + 15 = 65	**سمت چپ‌ترین جمع**
گام پنجم Y = 65 + 7;	
65 + 7 = 72	**آخرین جمع**
گام ششم Y = 72;	

شکل ۱۱-۲ | ترتیب اجرای عملگرهای محاسباتی در یک چند جمله‌ای درجه دوم.

عملگرهای مقایسه‌ای یا رابطه‌ای استاندارد در جبر	عملگرهای مقایسه‌ای یا رابطه‌ای در C++	مثال در C++	مفهوم شرط
عملگرهای مقایسه‌ای			
=	==	x ==y	x با y برابر است.
≠	!=	x!=y	x با y برابر نیست.
عملگرهای رابطه‌ای			
>	>	x>y	x از y بزرگتر است.
<	<	x<y	x از y کوچکتر است.
≥	>=	x>=y	x بزرگتر یا مساوی y است.
≤	<=	x<=y	x کوچکتر یا مساوی y است.

شکل ۱۲-۲ | عملگرهای مقایسه‌ای و رابطه‌ای.

خطای برنامه‌نویسی

در صورتیکه مابین هر کدامیک از عملگرهای ==، =!، =< و => فاصله قرار دهید با خطای نحوی مواجه خواهید شد.

خطای برنامه‌نویسی

معکوس نوشتن عملگرهای =!، =< و => بصورت !=، <= و >= خطای نحوی بدنبال خواهد داشت. در برخی از موارد نوشتن عملگر =! بصورت != خطای نحوی تلقی نمی‌شود اما بصورت یک خطای منطقی و در زمان اجرای برنامه تاثیر خود را نشان می‌دهد.

خطای برنامه‌نویسی

اشتباه گرفتن رفتار عملگر رابطه‌ای == با عملگر تخصیص = می‌تواند خطای منطقی بدنبال داشته باشد.

برنامه زیر از شش عبارت **if** برای مقایسه بین دو عدد ورودی از سوی کاربر استفاده می‌کند. اگر شرط موجود در هر کدامیک از عبارات **if** برقرار باشد (true)، خروجی مرتبط با آن عبارت به اجرا درخواهد آمد. برنامه شکل ۱۳-۲ نشاندهنده برنامه و کادرهای ورودی و خروجی از اجرای نمونه برنامه است.

```
1  // Fig. 2.13: fig02_13.cpp
2  // Comparing integers using if statements, relational operators
3  // and equality operators.
4  #include <iostream> // allows program to perform input and output
5
6  using std::cout; // program uses cout
7  using std::cin;  // program uses cin
8  using std::endl; // program uses endl
9
10 // function main begins program execution
11 int main()
12 {
13    int number1; // first integer to compare
14    int number2; // second integer to compare
15
16    cout << "Enter two integers to compare: "; // prompt user for data
17    cin >> number1 >> number2; // read two integers from user
18
19    if ( number1 == number2 )
20       cout << number1 << " == " << number2 << endl;
21
```

```
22    if ( number1 != number2 )
23       cout << number1 << " != " << number2 << endl;
24
25    if ( number1 < number2 )
26       cout << number1 << " < " << number2 << endl;
27
28    if ( number1 > number2 )
29       cout << number1 << " > " << number2 << endl;
30
31    if ( number1 <= number2 )
32       cout << number1 << " <= " << number2 << endl;
33
34    if ( number1 >= number2 )
35       cout << number1 << " >= " << number2 << endl;
36
37    return 0; // indicate that program ended successfully
38
39 } // end function main
```

```
Enter two integers to compare: 3 7
3 ! = 7
3 < 7
3 <= 7
```

```
Enter two integers to compare: 22 12
22 ! = 12
22 > 12
22 >= 12
```

```
Enter two integers to compare: 7 7
7 == 7
7 <= 7
7 >= 7
```

شکل ۱۳-۲ | عملگرهای رابطه‌ای و مقایسه‌ای.

در خطوط 6-8

```
using std::cout; // program uses cout
using std::cin;  // program uses cin
using std::endl; // program uses endl
```

از عبارات **using** استفاده شده است که نیاز به تکرار پیشوند **std::** را برطرف می‌کند. پس از بکارگیری عبارات **using**، می‌توانیم **cout** را بجای **std::cout**، **cin** را بجای **std::cin** و **endl** را بجای **std::endl** در مابقی برنامه بنویسیم.[توجه: از این نقطه به بعد در کتاب، در هر مثال از یک یا چند عبارت **using** استفاده شده است.]

برنامه‌نویسی ایده‌ال

ترجیحاً بلافاصله پس از #include از عبارت using استفاده کنید.

در خطوط 13-14

```
int number1; // first integer to compare
int number2; // second integer to compare
```

متغیرهای مورد نیاز برنامه اعلان شده‌اند. بخاطر دارید که متغیرها می‌توانند در یک خط یا چند خط اعلان شوند.

برنامه در خط 17 از روش آبشاری برای دریافت داده به منظور دریافت دو عدد صحیح استفاده کرده است. بخاطر دارید که امکان نوشتن **cin** را به توجه به خط 7 فراهم آورده‌ایم (بجای **std::cin**). اولین مقدار قرائت شده و در متغیر **number1** قرار داده می‌شود و سپس مقدار دوم قرائت شده و در متغیر **number2** ذخیره می‌گردد.

ساختار **if** در خطوط 19-20

```
if ( number1 == number2 )
      cout << number1 << " == " << number2 << endl;
```

مبادرت به مقایسه متغیرهای **number1** و **number2** برای تست برابر بودن می‌کند. اگر مقادیر برابر باشند، عبارت موجود در خط 20 جمله مبنی بر اینکه اعداد با هم برابر هستند به نمایش در می‌آورد. اگر شرطی در یک یا چند ساختار **if** که در خطوط 29,32 ,26 ,23 و 36 قرار دارند برقرار شود، عبارت متناظر توسط **cout** در خروجی به نمایش در می‌آید.

دقت کنید که هر ساختار **if** در برنامه شکل ۱۳-۲ دارای یک عبارت در بدنه خود بوده و بدنه تمام ساختارها بفرم دندانه‌دار نوشته شده‌اند. با دندانه‌دار نوشتن ساختار هر **if** وضوح و خوانائی برنامه را افزایش داده‌ایم. در فصل چهارم نشان خواهیم داد که چگونه می‌توان در ساختارهای **if** از چند عبارت استفاده کرد (به کمک جفت کاراکتر **{ }**).

برنامه‌نویسی ایده‌ال

دندانه‌دار نوشتن عبارت یا عبارات موجود در درون ساختار if سبب می‌شود تا بدنه ساختار بخوبی آشکار شده و بدنبال آن خوانائی برنامه افزایش یابد.

برنامه‌نویسی ایده‌ال

نبایستی بیش از یک عبارت در هر خط برنامه قرار دهید.

خطای برنامه‌نویسی

قرار دادن سیمکولن بلافاصله پس از شرط یک عبارت (پس از پرانتزها) ساختار if، خطای منطقی بدنبال خواهد داشت (اگرچه خطای نحوی به حساب نمی‌آید). سیمکولن سبب می‌شود، تا بدنه ساختار if خالی بنظر برسد، از اینرو ساختار if هیچ عملی انجام نمی‌دهد، صرفنظر از اینکه شرط برقرار باشد یا نباشد. علاوه بر این، بدنه اصلی ساختار if بصورت یک عبارت مجزا از if عمل می‌کند و همیشه اجرا شده و در اکثر مواقع نتایج اشتباه تولید می‌نماید.

به نحوه استفاده از فاصله‌ها در برنامه شکل ۱۳-۲ دقت کنید. در عبارات C++، کاراکترهای *white-space* همانند تب‌ها، خطوط جدید توسط کامپایلر در نظر گرفته نمی‌شوند. (اگر در درون رشته بکار گرفته شوند در نظر گرفته خواهند شد.) از اینرو، امکان دارد عبارات بر روی چند خط تقسیم شده و برطبق نظر برنامه‌نویس از هم فاصله پیدا کنند. جدا کردن هویت‌ها یا مشخصه‌ها، رشته‌ها (همانند "hello") و ثابت‌ها (همانند عدد 1000) بر روی چند خط خطای نحوی خواهد بود.

خطای برنامه‌نویسی

جدا کردن یک مشخصه از هم بوسیله کاراکترهای white-space خطای نحوی بدنبال خواهد داشت (برای مثال نوشتن main بصورت ma in).

برنامه‌نویسی ایده‌ال

می‌توانید یک عبارت طولانی را بر روی چند خط قرار دهید. اگر می‌بایست یک عبارت به چند خط تقسیم شود، نقطه تقسیم را از مکانی همانند ویرگول در لیست ویرگول‌ها، یا پس از یک عملگر در عبارات طولانی قرار دهید.

در جدول شکل ۱۴-۲ تقدم عملگرهای معرفی شده در این فصل بطور یکجا آورده شده‌اند. اولویت عملگرها از بالا به پایین کاهش می‌یابد. دقت کنید که تمام این عملگرها بجز از عملگر تخصیص =، از سمت چپ به راست ارزیابی می‌شوند.

عملگر	شرکت‌پذیری	نوع
( )	چپ به راست	پرانتز
* / %	چپ به راست	ضرب، تقسیم،باقیمانده
+ -	چپ به راست	جمع، تفریق
<< >>	چپ به راست	ورود و خروج داده
< <= > >=	چپ به راست	رابطه‌ای
== !=	چپ به راست	مقایسه‌ای
=	راست به چپ	تخصیصی

شکل ١٤-٢ تقدم عملگرهای معرفی شده تا بدین مرحله.

برنامه‌نویسی ایده‌ال

اگر عبارتی می‌نویسید که از عملگرهای متعددی تشکیل شده، بهتر است به جدول تقدم عملگرها مراجعه کنید. اگر از ترتیب عملگرها در عبارتی پیچیده مطمئن نیستید، از پرانتزها استفاده کرده و ترتیب اجرا را در دست گیرید.

۲-۸ مبحث آموزشی مهندسی نرم‌افزار: بررسی نیازمندی‌های ATM

در این بخش طراحی و پیاده‌سازی شی‌گرا، مبحث آموزشی مهندسی نرم‌افزار را آغاز می‌کنیم. بخش‌های «مبحث آموزشی مهندسی نرم‌افزار» که در انتهای این فصل و چند فصل بعدی قرار داده شده‌اند، شما را به آسانی وارد بحث شی‌گرایی خواهند کرد. نرم‌افزاری برای یک سیستم ماشین تحویل‌دار خودکار ATM ایجاد خواهیم کرد، که تجربه مناسبی در زمینه طراحی و پیاده‌سازی برایتان به ارمغان خواهد آورد.

در فصل‌های ۷-۳ و ۱۳، مراحل مختلفی از فرآیند طراحی شی‌گرا (OOD) را با استفاده از UML انجام خواهیم داد، و در کنار آن، مباحث مرتبط نیز در خود فصل‌ها مطرح می‌شوند. ضمیمه‌ای در ارتباط با پیاده‌سازی ATM با استفاده از تکنیک‌های برنامه‌نویسی شی‌گرا (OOP) در C++ آورده شده است. بحث ما یک بحث کاملاً آموزشی است، و حالت تمرینی ندارد و شما را کاملاً درگیر جزئیات کار با کد C++ می‌کند که پیاده‌سازی‌کننده برنامه هستند. این مطالب شما را با انواع مسائل قابل توجه در صنایع و همچنین راه‌حل‌های موجود آشنا خواهند کرد.

فرآیند طراحی را با معرفی مستند نیازمندی‌ها شروع می‌کنیم که تصریح‌کننده کل آنچیزی است که از یک سیستم ATM انتظار انجام آن را داریم و بطور دقیق آن را بررسی خواهیم کرد.

مستند نیازها

فرض کنید یک بانک محلی مایل است تا یک سیستم ATM جدید را بکار گیرد و به کاربران (مشتریان بانک) اجازه دهد تا تعاملات مالی خود را با آن انجام دهند (شکل ۱۵-۲). هر کاربر می‌تواند فقط یک حساب در بانک داشته باشد. کاربران ATM باید قادر به دیدن موجودی حساب، برداشت از حساب و پس‌انداز باشند.

واسط کاربر ATM حاوی کامپونت‌های سخت‌افزاری زیر است:

- یک صفحه نمایش که پیغام‌ها را به کاربر بنمایش درمی‌آورد
- یک صفحه کلید که ورودی عددی را از کاربر دریافت می‌نماید
- تحویل‌دار خودکار که پول را به کاربر تحویل می‌دهد
- شکاف سپرده که پاکت سپرده را از کاربر تحویل می‌گیرد.

تحویل‌دار خودکار هر روز با پانصد عدد 20 دلاری پر می‌شود. [نکته: به این علت که این مبحث آموزشی است، برخی از عناصر مشخص ATM توصیف شده در اینجا، دقیقاً مطابق با ATM واقعی نیستند. برای مثال، معمولاً در یک ATM واقعی دستگاهی وجود دارد که شماره حساب مشتری را از یک کارت ATM می‌خواند، در حالیکه در ATM ما از کاربر خواسته می‌شود که شماره حساب خود را از طریق صفحه کلید وارد کند. همچنین در یک ATM واقعی قبض رسید در پایان هر عملیات یا جلسه چاپ می‌شود. امام تمام خروجی‌ها در این ATM بر روی صفحه نمایش ظاهر می‌شوند.]

شکل ۱۵-۲ | واسط کاربر ATM.

بانک از شما می‌خواهد تا برنامه‌ای جدید برای انجام تعاملات مالی مشتریان بانک از طریق ATM توسعه دهید. بانک بعداً نرم‌افزار را با سخت‌افزار ATM مجتمع خواهد کرد. نرم‌افزار بایستی عملکرد دستگاه‌های سخت‌افزاری (همانند پرداخت‌کننده پول، دریافت‌کننده سپرده) را در درون کامپونت‌های نرم‌افزاری کپسوله کند، اما نیازی ندارد که از عملکرد داخلی و جزئیات آنها مطلع باشد. فعلاً بخش سخت‌افزاری ATM تولید نشده است، از اینرو بجای نوشتن نرم‌افزاری که بروی ATM اجرا شود، باید نسخه اولیه از نرم‌افزار را برای اجرا بر روی یک کامپیوتر شخصی ایجاد کنید. این نسخه از برنامه، از مانیتور کامپیوتر برای شبیه‌سازی صفحه نمایش ATM و صفحه کلید کامپیوتر برای شبیه‌سازی، صفحه کلید ATM استفاده خواهد کرد.

یک جلسه ATM متشکل از تصدیق یا تایید کاربر (اثبات هویت کاربر) برپایه شماره حساب و شماره شناسایی شخصی (PIN) بوده و در ادامه تعاملات مالی صورت می‌گیرد. برای تایید کاربر و انجام تعاملات، بایستی ATM با پایگاه داده اطلاعات حساب بانکی در تعامل قرار گیرد. [نکته: پایگاه داده یک مجموعه سازماندهی شده از اطلاعات ذخیره شده در یک کامپیوتر است.] برای هر حساب بانکی، پایگاه داده یک شماره حساب، یک PIN و یک موجودی که نشان‌دهنده مقدار پول در آن حساب است، در خود ذخیره می‌سازد. [نکته: برای ساده‌تر شدن کار، فرض می‌کنیم که هدف بانک فقط داشتن یک دستگاه ATM است، بنابراین نیازی نیست که نگران نحوه دسترسی همزمان چندین ATM به این پایگاه داده باشید. علاوه بر این، فرض می‌کنیم که بانک هیچ تغییری در زمان استفاده کاربر (مشتری) از ATM، در پایگاه داده اعمال نمی‌کند. همچنین هر سیستم تجاری همانند ATM به دلایل قابل قبولی در ارتباط با مباحث امنیتی است که خارج از قلمرو تحصیلی یک دانشجوی ترم اول یا دوم کامپیوتر است.]

در اولین برخورد، مشتری با ATM باید توالی از رویدادهای زیر رخ دهند (به شکل ۱۵-۲ توجه نمائید):

۱- صفحهٔ نمایش پیغام خوش‌آمدگویی (Welcome) را بنمایش درآورده و از کاربر می‌خواهد تا شماره حساب خود را وارد سازد.

۲- کاربر شماره حساب پنج‌رقمی خود را با استفاده از صفحه کلید وارد می‌سازد.

۳- در صفحه نمایش از کاربر خواسته می‌شود تا PIN را وارد سازد، که مرتبط با شماره حساب است.

۴- کاربر از طریق صفحه کلید، PIN پنج رقمی خود را وارد می‌سازد.

۵- اگر شماره حساب و PIN ورودی معتبر باشند، ظاهر صفحه نمایش همانند شکل ۱۶-۲ بوده و منوی اصلی در آن ظاهر می‌گردد. اگر شماره حساب یا PIN اشتباه باشد، پیغام مناسب در صفحه نمایش ظاهر شده و ATM به مرحله اول باز می‌گردد تا فرآیند تایید را از نو آغاز کند.

پس از تایید کاربرد از سوی ATM، منوی اصلی (شکل ۱۶-۲) لیستی از گزینه‌های عددی برای هر سه نوع تراکنش بنمایش در می‌آورد: درخواست موجودی (گزینه ۱)، برداشت (گزینه ۲) و سپرده‌گذاری (گزینه ۳). همچنین منوی اصلی، گزینه‌ای برای خروج از سیستم در اختیار کاربر قرار می‌دهد (گزینه ۴). پس از نمایش منوی اصلی، کاربر می‌تواند تراکنش موردنظر خود را از طریق وارد کردن 1، 2، 3 یا خروج از سیستم، 4 انتخاب کند. اگر کاربر گزینه اشتباهی را وارد سازد، پیغامی به نمایش درخواهد آمد، و سپس مجدداً منوی اصلی بنمایش در می‌آید.

اگر کاربر گزینهٔ 1 را انتخاب کند (با وارد کردن عدد 1) تا از میزان موجودی خود مطلع گردد، این امر صورت خواهد گرفت. برای انجام این کار، بایستی ATM میزان موجودی را از پایگاه داده بانک بازیابی کند.

مراحل زیر زمانی رخ می‌دهند که کاربر گزینه 2 را برای برداشت پول انتخاب کرده باشد:

۶- منوی در صفحه نمایش ظاهر می‌شود (شکل ۱۷-۲) که حاوی مقادیر استاندارد قابل پرداخت است. (گزینه 1) 20$، (گزینه 2) 40$، (گزینه 3) 60$، (گزینه 4)، 100$ و (گزینه 5) 200$. هم‌چنین این منو دارای یک گزینه برای لغو تراکنش کاربر است (گزینه 6).

شکل ۱۷-۲ | منوی برداشت پول ATM.

۷- کاربر با استفاده از صفحه کلید، انتخاب خود را انجام می‌دهد (گزینه‌های۶-۱).

۸- اگر مقدار درخواستی برای برداشت، بیشتر از میزان موجودی کاربر باشد، پیغامی این مطلب را به عرض کاربر رسانده و از وی می‌خواهد که مقدار کمتری تقاضا کند. سپس ATM به مرحله اول باز می‌گردد. اگر مقدار درخواستی کمتر از موجودی یا برابر آن باشد (یک مقدار قابل قبول)، ATM مرحله 4 را آغاز خواهد کرد. اگر کاربر مبادرت به لغو تراکنش کند (گزینه 6)، ATM منوی اصلی را به نمایش در آورده و منتظر ورودی کاربر می‌شود (شکل ۱۶-۲).

۹- اگر پرداخت‌کننده خودکار پول، به میزان کافی پول برای برآورده کردن تقاضای مشتری داشته باشد، ATM وارد مرحله ۵ خواهد شد. در غیر اینصورت، صفحه نمایش پیغامی مبنی بر اینکه میزان پول

دستگاه کمتر از مقدار درخواستی است از کاربر می‌خواهد که مقدار کمتری را انتخاب نماید. سپس ATM به مرحله اول بازمی‌گردد.

۱۰- ATM مقدار پول برداشتی را از موجودی حساب کاربر در پایگاه داده بانک کم می‌کند.

۱۱- پرداخت‌کننده خودکار، مقدار پول درخواستی را به کاربر تحویل می‌دهد.

۱۲- پیغامی در صفحه نمایش ظاهر شده و به کاربر یادآوری می‌کند، پول را بردارد.

مراحل زیر زمانی رخ می‌دهند که کاربر عدد 3 را از منوی اصلی به منظور سپرده‌گذاری انتخاب کرده باشد:

۱۳- در صفحه نمایش، به کاربر اعلان می‌شود که مقدار سپرده‌گذاری خود را وارد سازد یا برای لغو تراکنش صفر را وارد کند.

۱۴- کاربر از طریق صفحه کلید، مقدار سپرده‌گذاری یا صفر را وارد می‌سازد [نکته: صفحه کلید دارای نقطه اعشار یا نماد دلار نمی‌باشد، از اینرو، کاربر نمی‌تواند یک مقدار دلاری واقعی همانند $1.25 را وارد سازد. بجای آن، کاربر باید مقدار سپرده خود را بعنوان یک عدد از سنت‌ها وارد کند (مثلاً 125). سپس ATM این عدد را بر 100 تقسیم می‌کند، تا عددی که نشاندهنده مقدار دلاری است بدست آید (مثلاً 125÷100=1.25)].

۱۵- اگر کاربر میزان سپرده را مشخص سازد، ATM به مرحله ۴ می‌رود. اگر کاربر، مبادرت به لغو تراکنش کند (با وارد کردن صفر)، ATM منوی اصلی را به نمایش درآورده و در انتظار ورودی کاربر باقی می‌ماند (شکل ۱۶-۲)

۱۶- در صفحهٔ نمایش، پیغامی به کاربر اعلان می‌کند که پاکت سپرده را در شکاف سپرده قرار دهد.

۱۷- اگر شکاف سپرده، پاکت سپرده را در عرض دو دقیقه دریافت نماید، ATM مبادرت به افزایش اعتبار کاربر، در میزان موجودی وی در پایگاه داده بانک می‌کند [نکته: این مقدار پول، بلافاصله برداشت نمی‌شود. ابتدا باید بانک به لحاظ فیزیکی مبادرت به بازبینی پول نقد و چک‌های موجود در پاکت کرده و پس از تایید بانک، حساب کاربر در پایگاه داده به روز می‌شود. این عملیات مستقل از سیستم ATM صورت می‌گیرد.] اگر در مدت زمان مشخص شده، شکاف سپرده، پاکتی دریافت نکند، پیغامی در صفحه نمایش مبنی بر لغو تراکنش از طرف سیستم ظاهر خواهد شد. سپس ATM منوی اصلی را به نمایش درآورده و منتظر ورودی کاربر باقی ماند.

پس از اجرای موفقیت‌آمیز یک تراکنش توسط سیستم، باید مجدداً منوی اصلی بنمایش درآید (شکل ۱۶-۲) تا کاربر قادر به انجام تراکنش‌های دیگر باشد. اگر کاربر گزینه ۴ را انتخاب کند (خروج از سیستم)، پیغام تشکر بنمایش درآمده و سپس پیغام خوش‌آمدگویی، برای کاربر بعدی بنمایش درمی‌آید.

تحلیل سیستم ATM

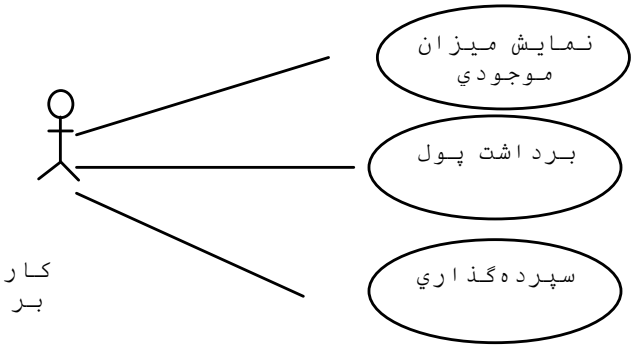
بخش قبلی مثال ساده شده‌ای از یک مستند نیازمندی‌ها بود. بطور نمونه، چنین مستندی، نتیجه‌ای از نیازمندی‌های جمع‌آوری شده است که شامل مصاحبه با کاربران اصلی سیستم و متخصصین در فیلدهای مربوط به سیستم است. برای مثال، تحلیل‌گر سیستم کسی است که برای آماده کردن مستند نیازمندی‌های نرم‌افزار بانک استخدام می‌شود (مثلاً سیستم ATM که در اینجا توضیح داده شده است) و می‌تواند با متخصصین امور مالی مصاحبه انجام دهد تا درک دقیقی از کاری که نرم‌افزار باید انجام دهد بدست آورد. تحلیل‌گر از اطلاعات بدست آمده، لیستی از نیازهای سیستم جمع‌آوری می‌کند تا طراحان سیستم را بخوبی راهنمایی کند.

فرآیند جمع‌آوری اطلاعات نیازمندی‌ها، یک وظیفه کلیدی در مرحله اول چرخه زندگی نرم‌افزار است. چرخه زندگی نرم‌افزار، تصریح‌کننده مراحلی است که نرم‌افزار از بدو تولد تا زمان بازنشستگی طی می‌کند. بطور نمونه این مراحل عبارتند از: تحلیل، طراحی، پیاده‌سازی، تست و خطایابی، استفاده، نگهداری و بازنشستگی. چندین مدل برای چرخه طول عمر نرم‌افزار وجود دارد که هر یک دارای مزایا و مشخصات خاص بوده که روش انجام مراحل مختلف را به مهندسان نرم‌افزار گوشزد می‌کنند. مدل آبشاری (waterfall model) هر مرحله را یکی بعد از دیگری انجام می‌دهد، در حالیکه مدل تکرارکننده (iterative model) می‌تواند مراحل را یک یا چندین بار در فرآیند چرخه طول عمر یک محصول تکرار نماید.

مرحله تحلیل در چرخه طول عمر نرم‌افزار، متمرکز بر تعریف مسئله برای حل کردن آن است. به هنگام طراحی هر سیستمی، باید مسئله بدرستی حل شود (راه حل باید صحیح باشد). تحلیل‌گران سیستم مبادرت به جمع‌آوری نیازهایی می‌کنند که قادر به حل مسئله مشخصی هستند. مستند نیازها که آن را در اینجا برای سیستم ATM مطرح کرده‌ایم، بقدر کافی گویا است و نیازی نیست که وارد یک مرحله تحلیل اضافی‌تر شوید.

برای ثبت اینکه سیستم مورد نظر چه کاری باید انجام دهد، غالباً توسعه‌دهندگان از تکنیکی بنام مدل‌سازی use case (حالت استفاده) کمک می‌گیرند. این فرآیند حالات مورد استفاده از سیستم را معین می‌سازد، که هر یک نشانگر یک قابلیت مختلف است که سیستم در اختیار سرویس‌گیرندگان خود قرار

می‌دهد. برای مثال، بطور نمونه ATMها دارای چندین حالت استفاده همانند «نمایش میزان موجودی»، «برداشت پول»، «سپرده‌گذاری»، «انتقال پول مابین حساب‌ها» و «خرید تمبر پستی» هستند. سیستم ATM ساده شده که قصد ساخت آن را داریم، فقط دارای سه حالت استفاده است که در شکل ۱۸-۲ دیده می‌شود.



شکل ۱۸-۲ | دیاگرم حالت استفاده برای سیستم ATM از منظر کاربر.

هر حالت استفاده توصیف‌کننده یک سناریو است که کاربر از سیستم استفاده می‌کند. در حال حاضر حالات استفاده از سیستم ATM را از مستند نیازها مطالعه کرده‌اید، که هر مرحله نیازمند نوعی از تراکنش است (نمایش موجودی، برداشت پول و سپرده‌گذاری)، که آنها را در سه حالت استفاده از ATM قرار داده‌ایم.

دیاگرام‌های Use Case

در این بخش مبادرت به معرفی یکی از چندین دیاگرام UML برای ATM مطرح شده در این مبحث آموزشی می‌کنیم. یک دیاگرام حالت استفاده (use case) برای مدل کردن تراکنش‌های مابین سرویس‌گیرندگان سیستم (در این مورد، مشتریان بانک مدنظر هستند) و سیستم ایجاد می‌کنیم. هدف، نمایش انواع تراکنش‌های کاربران با سیستم است بدون اینکه جزئیات را وارد کار نمائیم (جزئیات در دیاگرام‌های دیگر UML تدارک دیده می‌شوند، که در ادامه با آنها مواجه خواهید شد). غالباً دیاگرام‌های حالت استفاده همراه با جملات غیررسمی که توصیف‌کننده دقیق‌تر حالات استفاده هستند، همراه می‌باشند، همانند جملاتی که در مستند نیازها دیده می‌شود. دیاگرام‌های حالت استفاده در مرحله تحلیل چرخه عمر نرم‌افزار تولید می‌شوند. در سیستم‌های بزرگتر، دیاگرام‌های حالت استفاده ساده هستند اما جزء

ابزارهای ضروری هستند که به طراحان سیستم کمک می‌کنند تا تمرکز خود را بر روی نیازهای کاربران حفظ کنند.

شکل ۱۸-۲ نمایشی از دیاگرام حالت استفاده برای سیستم ATM است. تصویر آدمک نشان‌دهنده یک *بازیگر* (*actor*) است که تعریف‌کننده نقش‌های است که یک موجودیت خارجی همانند یک شخص یا سیستم دیگر، به هنگام در تعامل قرار گرفتن با سیستم ایفاء می‌کند. برای سیستم ATM ما، بازیگر کاربری است که می‌تواند میزان موجودی را مشاهده کند، از سیستم پول دریافت و در آن سپرده‌گذاری انجام دهد. کاربر یک شخص حقیقی نیست، اما دارای نقش‌های از یک شخص واقعی است (زمانیکه نقشی از یک کاربر را باز می‌کند) و می‌تواند آن نقش‌ها را در زمان تعامل با ATM بازی‌کند. دقت کنید که دیاگرام حالت استفاده می‌تواند حاوی چندین بازیگر باشد. برای مثال، دیاگرام حالت استفاده در یک سیستم ATM واقعی، می‌تواند شامل یک بازیگر بنام Administrator (مدیر) باشد که مسئول پر کردن هر روز پول در تحویل‌دار است.

شناسایی بازیگر در سیستم را با بررسی مستند نیازها مشخص می‌کنیم که عبارت است از "کاربران ATM بایستی قادر به مشاهده میزان موجودی، برداشت پول و سپرده‌گذاری باشند." از اینرو، بازیگر در هر یک از این سه حالت استفاده، کاربری است که با ATM در تعامل قرار می‌گیرد. موجودیت خارجی، یک شخص حقیقی، بخشی از نقش کاربر را در انجام تراکنش‌های مالی بازی می‌کند. در شکل ۱۸-۲ یک بازیگر بنام «کاربر» نشان داده شده است. UML هر کدامیک از حالات استفاده را بصورت یک بیضی متصل به بازیگر توسط یک خط ساده را مدل می‌کند.

بایستی مهندسان نرم‌افزار یا بطور دقیق‌تر طراحان سیستم، مبادرت به تحلیل مستند نیازها یا تنظیم حالات استفاده و طراحی سیستم قبل از شروع به برنامه‌نویسی با یک زبان برنامه‌نویسی خاص کنند. در مدت زمان مرحله تحلیل، طراحان سیستم بر درک مستند نیازها تمرکز دارند تا مشخصاتی با معیار بالا بدست آید که توصیف‌کننده آنچه که سیستم باید انجام دهد، باشد. خروجی مرحله طراحی (طراحی مشخصه‌ها) بایستی بقدر کافی واضح و شفاف باشد که نحوه ایجاد سیستم را برای برآورده کردن نیازها بیان کند. در چند بخش بعدی «مبحث آموزشی مهندسی نرم‌افزار» این مراحل را از طریق طراحی شی‌گرا (OOD) بر روی سیستم ATM به منظور تولید طرحی از مشخصه‌ها انجام خواهیم داد که حاوی مجموعه‌ای از دیاگرام‌های UML و جملات پشتیبانی‌کننده است. بخاطر دارید که UML برای استفاده در هر فرآیند OOD طراحی شده است. چندین پردازش‌کننده وجود دارد که یکی از بهترین آنها RUP (Rational Unified Process) است که توسط شرکت Rational Software توسعه پیدا کرده است (این

شرکت تبدیل به بخشی از IBM شده است). RUP یک پردازش‌کننده بسیار مناسب و مطلوب برای طراحی برنامه‌های کاربردی در سطح صنایع است. برای این مبحث آموزشی، فرآیند طراحی ساده شده خود را معرفی می‌کنیم.

طراحی سیستم ATM

در این بخش وارد مرحله طراحی سیستم ATM می‌شویم. سیستم مجموعه‌ای از کامپونت‌ها است که برای حل مسئله‌ای با هم در تعامل قرار می‌گیرند. برای مثال، برای اینکه سیستم ATM وظایف تعیین شده را انجام دهد، باید دارای یک واسط کاربر بوده (شکل ۱۵-۲) و حاوی نرم‌افزاری باشد که قادر به انجام تراکنش‌های مالی و کار با پایگاه داده اطلاعات حساب مشتریان در بانک باشد. ساختار سیستم، توصیف‌کننده شی‌های سیستم و روابط داخلی آنها است. رفتار سیستم توصیف‌کننده نحوه تغییر عملکرد شی‌های سیستم و برقراری روابط بین آنها است. هر سیستمی دارای ساختار و رفتار است، که طراحان باید آنها را مشخص سازند. چندین نوع مشخص از ساختار و رفتار سیستم وجود دارد. برای مثال، تعامل مابین شی‌ها در یک سیستم، متفاوت از تعامل مابین سیستم و کاربر است، با این همه هنوز هر دو بخشی از رفتار سیستم محسوب می‌شوند. نسخه UML 2 تصریح‌کننده 13 نوع دیاگرام برای مستند کردن مدل‌های سیستم است. هر دیاگرام صفات مشخصی از ساختار سیستم یا رفتار آن را مدل‌سازی می‌کند، شش دیاگرام در ارتباط با ساختار سیستم بوده و هفت دیاگرام باقیمانده مربوط به رفتار سیستم هستند. در اینجا فقط شش نوع دیاگرام بکار رفته در این مبحث را لیست کرده‌ایم، یکی از آنها بنام دیاگرام کلاس، مبادرت به مدل کردن ساختار سیستم می‌کند و مابقی در ارتباط با مدل‌سازی رفتار سیستم هستند.

۱- دیاگرام حالت استفاده، همانند شکل ۱۸-۲، مبادرت به مدل‌سازی تعامل صورت گرفته مابین سیستم و موجودیت خارجی آن (بازیگران) با جملات حالت استفاده می‌کند (قابلیت‌های سیستم همانند «نمایش میزان موجودی»، «برداشت پول» و «سپرده‌گذاری»).

۲- دیاگرام‌های کلاس، که در بخش ۱۱-۳ با آنها آشنا خواهید شد. این دیاگرام‌ها در مدل کردن کلاس‌ها یا «ایجاد بلوک‌های» مورد استفاده در سیستم کاربرد دارند. هر اسم یا «چیز» توصیف شده در مستند نیازها، نامزد تبدیل شدن به یک کلاس در سیستم است (همانند «حساب»، «صفحه کلید»). دیاگرام‌های کلاس در مشخص کردن روابط ساختاری موجود مابین اجزای سیستم نقش دارند. برای مثال، دیاگرام کلاس سیستم ATM مشخص می‌کند که ATM به لحاظ فیزیکی متشکل از یک صفحه نمایش، صفحه کلید، تحویل‌دار خودکار پول و شکاف سپرده‌گذاری است.

۳- دیاگرام‌های وضعیت ماشین، که در بخش ۱۱-۳ با آنها آشنا خواهید شد. این دیاگرام‌ها در مدل‌سازی روش‌های که یک شی تغییر وضعیت یا حالت می‌دهد کاربرد دارند. وضعیت یک شی توسط مقادیری که از صفات شی در زمان اجرا بدست می‌آیند، تعیین می‌شود. زمانیکه وضعیت یک شی تغییر پیدا می‌کند، امکان دارد شی رفتار متفاوتی در سیستم بخود بگیرد. برای مثال، پس از اعتبارسنجی PIN کاربر، تراکنش ATM از وضعیت «کاربر تایید نشده» به وضعیت «کاربر تایید شده» تبدیل شده و در این لحظه ATM به کاربر اجازه می‌دهد تا تراکنش‌های مالی انجام دهد (مشاهده میزان موجودی، برداشت پول و سپرده‌گذاری).

۴- دیاگرام‌های فعالیت، که در بخش ۱۱-۵ با آنها آشنا خواهید شد. این دیاگرام‌ها در مدل‌سازی فعالیت یک شی کاربرد دارند (جریان کار یا روند کار یک شی (توالی از رویدادها) در مدت زمان اجرای برنامه). یک دیاگرام فعالیت مبادرت به مدل کردن اعمال یک شی و همچنین ترتیب انجام این اعمال را مشخص می‌نماید. برای مثال، یک دیاگرام فعالیت نشان می‌دهد که ATM باید میزان موجودی حساب کاربر را (از پایگاه داده اطلاعات حساب) قبل از اینکه صفحه نمایش موجودی را بنمایش درآورد، تهیه نماید.

۵- دیاگرام‌های ارتباطی (در نسخه‌های قبلی UML این دیاگرام، دیاگرام‌های همکاری نامیده می‌شود) مدل‌کننده تعامل‌های صورت گرفته مابین شی‌های یک سیستم هستند، با تاکید بر اینکه کدام تعامل رخ دهد. در بخش ۱۲-۷ با این نوع دیاگرام‌ها آشنا خواهید. برای مثال باید ATM با پایگاه داده حساب بانکی ارتباط برقرار کند تا میزان موجودی را بازیابی نماید.

۶- دیاگرام‌های توالی، این دیاگرام‌ها نیز مبادرت به مدل‌سازی تعامل‌های صورت گرفته مابین شی‌های یک سیستم می‌کنند، اما برخلاف دیاگرام‌های ارتباطی، تاکید آنها بر زمان رخ دادن تعامل‌ها است. در بخش ۱۲-۷ با این نوع دیاگرام‌ها آشنا خواهید شد. برای مثال، صفحه نمایش به کاربر اطلاع می‌دهد که مقدار پولی که می‌خواهد برداشت کند را قبل از پرداخت وارد نماید.

در بخش ۱۱-۳ به ادامه طراحی ATM با شناسایی کلاس‌ها از طریق مستند نیازها ادامه خواهیم داد. این کار را با استخراج اسامی کلیدی و تعبیر اسامی از مستند نیازها انجام خواهیم داد. با استفاده از این کلاس‌ها، اولین پیش‌نویس خود را از دیاگرام کلاس ایجاد می‌کنیم که ساختار سیستم ATM را مدل‌سازی می‌کند.

اینترنت و منابع وب

www.306.ibm.com/software/rational/uml/

www.softdocwiz.com/Dictonary.htm
www-306.ibm.com/software/rational/offerings/design.html
www.embarcadero.com/products/describe/index.html
www.borland.com/together/index.html
www.ilogix.com/rhapsody/rhapsody.cfm
argouml.tigris.org
www.objectsbydesign.com/booklist.html
www.objectsbydesign.com/tools/umltools-bycompany.html
www.ootips.org/ood-principles.html
www.cetus-link.org100-uml.html
www.agilemodeling.com/essays/umlDiagrams.htm

کتاب‌های توصیه شده

کتاب‌های معرفی شده در این بخش حاوی اطلاعاتی در ارتباط با طراحی شی‌گرا با UML هستند.

- Booch,G. Object-Oriented Analysis and Design with Applications, Third Edition. Addision-Wesly,2004.
- Eriksson,H.,et al. UML2 Toolkit.NewYork:John Wiley,2003.
- Kruchten,P.The Rational Unified Process:An Introduction.Boston: Addison-Wesley, 2004.
- Lorman,C.Applying UML and Patterns:An Introduction to object. Oriented Analysis and Design, Second Edition.Upper Saddle River, NT:Prentice Hall, 2002.
- Roques,P.UML in Practical:The Art of Modeling Software Systems Demonstrated through worked Examples and solutions.NewYork:John wiley,2004.
- Rosenberg,D.,and K.Scott.Applying use Case Driven Object Modeling withUML:An Anstated e-Commerce Example.Reading,MA:Addison-Wesley, 2001.
- Rumbaugh,J.,I.Jacobson and G.Booch.The Complete UML Training Course.Upper Saddle River,NS:Prentice Hall,2000.
- Rumbaugh,J.,I.Jacobson and G. Booch. The unified Modeling Language Reference Manual. Reading, MA: Addison-Wesley,1999.
- Rumbaugh,J.,I.Jacobson an g.Booch.The unified Software Development Process. Reading, MA: Addison-Wesley,1999.

خودآزمایی مبحث آموزشی مهندسی نرم‌افزار

۲-۱ فرض کنید که سیستم ATM موردنظر ما قادر به انتقال پول بین دو حساب بانکی است. در دیاگرام حالت استفاده شکل ۱۸-۲ این تغییر را اعمال کنید.

۲-۲ _____، مدل‌کننده تعامل مابین شی‌ها در یک سیستم با تاکید بر زمان رخ دادن تعامل‌ها است.

a) دیاگرام‌های کلاس

b) دیاگرام‌های توالی

c) دیاگرام‌های ارتباطی

d) دیاگرام‌های فعالیت

۲-۳ کدامیک از لیست‌های زیر نشاندهنده چرخه عمر صحیح نرم‌افزار هستند؟

a) طراحی، تحلیل، پیاده‌سازی، تست

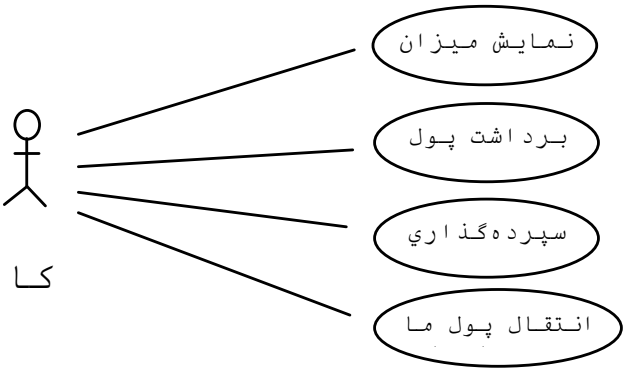
b) طراحی، تحلیل، تست، پیاده‌سازی

c) تحلیل، طراحی، تست، پیاده‌سازی

d) تحلیل، طراحی، پیاده‌سازی، تست

پاسخ خودآزمایی مبحث آموزشی مهندسی نرم‌افزار

۲-۱ شکل ۱۹-۲ حاوی دیاگرام حالت استفاده اصلاح شده از سیستم ATM است که به کاربر امکان انتقال پول مابین حساب‌ها را فراهم می‌آورد.



شکل ۱۹-۲ | دیاگرام حالت برای نسخه‌ای از ATM که قادر به انجام انتقال پول مابین حساب‌ها هم می‌باشد.

b **۲-۲**

d **۲-۳**

خودآزمایی

۲-۱ جاهای خالی را در عبارات زیر با کلمات مناسب پر کنید:

a) هر برنامه C++ اجرای خود را از تابع ..... آغاز می‌کند.

b) بدنه هر تابع با کاراکتر ..... شروع و با کاراکتر ..... به پایان می‌رسد.

c) هر عبارتی با کاراکتر ..... به پایان می‌رسد.

d) کاراکتر توالی **\n** نشاندهنده کاراکتر ..... است، که باعث می‌شود تا کرسر به ابتدای خط بعدی در صفحه نمایش منتقل شود.

e) از عبارت ..... برای تصمیم‌گیری استفاده می‌شود.

۲-۲ کدامیک ازعبارات زیر صحیح و کدامیک اشتباه است. اگر عبارتی اشتباه است علت آنرا توضیح دهید. فرض کنید از عبارت **using std::cout;** استفاده شده است.

a) توضیحات سبب می‌شوند تا کامپیوتر مبادرت به چاپ عبارت قرار گرفته پس از // بر روی صفحه نمایش به هنگام اجرای برنامه کند.

b) کاراکتر توالی **n** به هنگام کار با **cout** موجب می‌شود تا کرسر به ابتدای خط بعدی در صفحه نمایش منتقل شود.

c) قبل از اینکه بتوان از متغیری استفاده کرد باید آن را اعلان کرده باشیم.

d) به هنگام اعلان متغیرها بایستی نوع آنها تعریف شود.

e) در نظر C++ متغیرهای **number** و **NuMbEr** یکسان هستند.

f) تقریبا می‌توان اعلان‌ها را در هر کجای بدنه یک تابع C++ قرار داد.

g) از عملگر **%** فقط می‌توان در کنار عملوندهای صحیح استفاده کرد.

h) تمام عملگرهای محاسباتی + , **%** , / , * و - دارای اولویت برابر هستند.

i) یک برنامه C++ که سه خط در خروجی چاپ می‌کند بایستی حاوی سه عبارت خروجی با بکارگیری **cout** و عملگر درج باشد.

۳-۲ یک عبارت تک جمله‌ای در C++ بنویسید که موارد مورد تقاضا را برآورده سازد: (فرض کنید که از عبارت **using** استفاده نشده است)

a) متغیرهای **q76354** ,**thisIsAvariable** ,c و **number** را از نوع **int** اعلان کنید.

b) به کاربر اعلان کنید تا یک عدد صحیح وارد کند. در انتهای پیغام یک کاراکتر کولن (:) و سپس یک فاصله قرار دهید.

c) مقدار صحیح وارد شده از طریق صفحه کلید را دریافت و آنرا در متغیر **age** ذخیره سازد.

d) اگر متغیر **number** برابر 7 نباشد، عبارت "The variable number is not equal to 7" چاپ شود.

e) پیغام "This is a C++ program" در یک خط چاپ شود.

f) پیغام "This is a C++ program" را دو خط چاپ کند. بطوریکه خط اول با C++ به پایان برسد.

g) پیغام "This is a C++ program" را به نحوی چاپ کند، که هر کلمه در خط مجزائی چاپ شود.

h) پیغام "This is a C++ program" را به نحوی چاپ کند، که هر کلمه با کلمه دیگر به اندازه یک tab فاصله داشته باشد.

٤-۲ یک عبارت (یا توضیح) برای برآورده کردن موارد زیر بنویسید (فرض کنید که از عبارت **using** استفاده شده است):

a) نشان دهد که برنامه مبادرت به ضرب سه عدد صحیح می‌کند.

b) متغیرهای **x, y, z** و **result** از نوع **int** اعلان شوند (در عبارات مجزا).

c) به کاربر اعلان شود تا سه عدد صحیح وارد برنامه سازد.

d) سه مقدار صحیح از صفحه‌کلید دریافت آنها را در متغیرهای **x, y** و **z** ذخیره کند.

e) حاصلضرب سه مقدار موجود در متغیرهای **x, y** و **z** را بدست آورده و آنرا در متغیر **result** ذخیره کند.

f) عبارت "The product is" را قبل از مقدار متغیر **result** چاپ کند.

g) مقداری از **main** باز گرداند، تا نشان دهد برنامه با موفقیت پایان پذیرفته است.

۲-۵ با استفاده از عبارات نوشته شده در تمرین ۴-۲، یک برنامه کامل بنویسید که حاصلضرب سه مقدار صحیح را بدست آورده و نتیجه را به نمایش در آورد. [توجه: از عبارات **using** استفاده کنید.]

۲-٦ خطاهای موجود در عبارات زیر را تشخیص داده و آنها را اصلاح کنید (فرض کنید که از عبارت **using std::cout;** استفاده شده است):

```
a) if ( c < 7 ) ;
     cout << "c is less than 7\n";
b) if ( c => 7 )
     cout << "c is equal to or greater than 7\n";
```

پاسخ خودآزمایی

۲-۱ a) **main** b) براکت چپ {، براکت راست }. c) سیمکولن. d) خط جدید. e) **if**

۲-۲ a) اشتباه. توضیحات هیچ عملی به هنگام اجرای برنامه انجام نمی‌دهند. از آنها فقط برای مستند کردن و افزایش خوانائی برنامه استفاده می‌شود.

b) صحیح.

c) صحیح.

d) صحیح.

e) اشتباه. زبان C++ حساس به موضوع است، از اینرو متغیرها با یکدیگر فرق دارند.

f) صحیح.

g) صحیح.

h) اشتباه. عملگرهای / ,* و % دارای اولویت یکسان بوده و عملگرهای + و – از اولویت پایین‌تر برخوردار هستند.

i) اشتباه. یک عبارت خروجی با استفاده از **cout** حاوی چندین توالی **\n** می‌تواند در چندین خط چاپ شود.

۲-۳

```
a) int c, thisIsVariable, q76354, number;
b) std::cout << "Enter an integer: " ;
c) std::cin >> age;
d) if (number != 7 )
     std::cout << "The variable number is not equal to 7 \n";
e) std::cout << "This is a C++ program\n";
f) std::cout << "This is a C++\nprogram\n";
g) std::cout << "This\nis\na\nC++\nprogram\n";
h) std::cout << "This \tis\ta\tC++\tprogram\n";
```

۲-٤

```
a) // Calculate the product of three integers
b) int x;
```

```
   int y;
   int z;
   int result;
```

c) `cout << "Enter three integers: ";`

d) `cin >> x >> y >> z;`

e) `result = x * y * z;`

f) `cout << "The product is " << result << endl;`

g) `return 0;`

۲-۵

```
1       // Calculate the product of three integers
2       #include <iostream>
3
4       using std::cout;
5       using std::cin;
6       using std::endl;
7
8       //function main begins program execution
9       int main()
10      {
11         int x; //first integer to multiply
12         int y; //second integer to multiply
13         int z; //third integeer to multiply
14         int result; //the product of the three integer
15
16         cout << "Enter three integers: "; //prompt user for data
17         cin >> x >> y >> z; //read three integers from user
18         result = x * y * z; //multiply the three integers; store result
19         cout << "The product is " <<result<<endl;//print result; end line
20
21         return 0; //indicate program executed successfully
22      }  // end function main
```

۲-۶ a) خطا. سیمکولن پس از پرانتز در سمت راست قرار گرفته است. برای اصلاح این خطا بایستی سیمکولن پس از پرانتز سمت راست را حذف کنید.[توجه: در نتیجه اجرای این قسمت از برنامه با خطای موجود، عبارت پس از شرط **if** در هر حالتی به اجرا در خواهد آمد.]

b) خطا: عملگر رابطه‌ای بصورت< = تایپ شده است. برای اصلاح این خطا، عملگر باید بفرم =< تایپ شود.

تمرینات

۲-۷ در ارتباط با هر کدامیک از شی‌های زیر توضیح دهید:

a) `std::cin`

b) `std::cout`

۲-۸ جاهای خالی در عبارات زیر را با کلمات مناسب پر کنید:

a) ..... در افزایش خوانائی و مستند کردن یک برنامه نقش دارند.

b) از شی ..... برای چاپ اطلاعات بر روی صفحه نمایش استفاده می‌شود.

c) عبارت ..... در تصمیم‌گیری بکار گرفته می‌شود.

d) معمولا محاسبات توسط عبارات ..... انجام می‌شوند.

e) شی ..... مقادیر را از صفحه‌کلید دریافت می‌کند.

۲-۹ یک عبارت تک جمله‌ای در C++ بنویسید که موارد مورد تقاضا را برآورده سازد:

a) جمله "Enter two numbers" چاپ شود.

b) حاصلضرب متغیرهای **b** و **c** در متغیر **a** ذخیره شود.

c) مشخص کنید که برنامه عمل محاسبه پرداخت دستمزد را انجام می‌دهد (عبارت توضیحی بنویسید).

d) سه مقدار صحیح از صفحه‌کلید دریافت و در متغیرهای صحیح **a** , **b** و **c** قرار دهد.

۲-۱۰ کدام عبارات صحیح و کدام اشتباه است. اگر عبارتی اشتباه است علت آنرا توضیح دهید.

a) ارزیابی عملگرهای C++ از سمت چپ به راست صورت می گیرد.

b) تمام اسامی متغیر زیر معتبر هستند:

_under_bar_ , m928134, t5, j7, her_sales, his_account_total, a, b, c, z, z2

c) عبارت **cout << "a = 5;"**، مثالی از یک عبارت تخصیصی است.

d) یک عبارت محاسباتی معتبر C++ بدون وجود پرانتز از سمت چپ به راست ارزیابی می‌شود.

e) تمام اسامی متغیر زیر معتبر نمی‌باشند:

3g, 87, 67h2, h22, 2h

۲-۱۱ جاهای خالی در عبارات زیر را با کلمات مناسب پر کنید:

a) کدام عملیات محاسباتی در سطح یکسانی از تقدم ضرب قرار دارد؟.....

b) زمانیکه پرانتزها بصورت تودرتو هستند، کدام جفت پرانتز ابتدا ارزیابی می‌شود؟.....

c) مکانی در حافظه کامپیوتر که می‌تواند در هر بار مقدار متفاوتی داشته باشد، ..... نامیده می‌شود.

۲-۱۲ با انجام عبارات زیر چه اتفاقی رخ می‌دهد. با فرض **x = 2** و **y = 3**

a) `cout << x;`
b) `cout <<  x + x;`
c) `cout << "x=";`
d) `cout << "x = " << x;`
e) `cout << x + y << " = " << y + x;`
f) `z = x + y;`
g) `cin >> x >> y;`
h) `//cout << "x + y = " << x + y;`
i) `cout << "\n";`

۲-۱۳ کدام یک از عبارات C++ زیر حاوی متغیرهای هستند که مقادیر آنها تغییر خواهد یافت؟

a) `cin >> b >> c >> d >> e >> f;`
b) `p = i + j + k + 7;`
c) `cout << "variables whose values are replaced"`
d) `cout << "a = 5";`

۲-۱٤ با توجه به معادله $y = ax^3 + 7$ کدامیک از عبارات زیر پاسخ صحیح این معادله هستند؟

a) `y = a * x * x * x + 7;`
b) `y = a * x * x * ( x + 7 );`
c) `y = ( a * x ) * x * ( x + 7 );`
d) `y = (a * x ) * x * x + 7;`
e) `y = a * ( x * x * x ) + 7;`

f) `y = a * x * ( x * x + 7 );`

۲-۱۵ ترتیب ارزیابی عملگرهای زیر را در عبارات C++ مشخص سازید و مقدار x را بدست آورید.

a) `x = 7 + 3 * 6 / 2 -1;`
b) `x = 2 % 2 + 2 * 2 -2 / 2;`
c) `x = ( 3 * 9 * ( 3 + ( 9 * 3 / ( 3 ) ) ) );`

۲-۱۶ برنامه‌ای بنویسید که دو عدد از کاربر دریافت و سپس مجموع، ضرب، تفریق و تقسیم آنها را بدست آورده و چاپ کند.

۲-۱۷ برنامه‌ای بنویسید که اعداد 1 تا 4 را بر روی یک خط چاپ کند، به نحویکه مابین هر جفت یک کاراکتر فاصله قرار گرفته باشد. برنامه را با استفاده از روش‌های زیر بنویسید:

a) با استفاده از یک عبارت خروجی با یک عملگر درج.

b) با استفاده از یک عبارت خروجی با چهار عملگر درج.

c) با استفاده از چهار عبارت.

۲-۱۸ برنامه‌ای بنویسید که از کاربر تقاضای دریافت دو عدد صحیح کرده و آنها را دریافت و سپس عدد بزرگتر را قبل از جمله "is larger" چاپ کند. اگر دو عدد با هم برابر باشند، پیغام "These numbers are equal" چاپ شود.

۲-۱۹ برنامه‌ای بنویسید که از طریق صفحه‌کلید سه عدد صحیح دریافت کرده و سپس مجموع، میانگین، حاصلضرب، کوچکترین و بزرگترین عدد را چاپ کند. خروجی بایستی بفرم زیر طراحی شود:

```
Input three different integers: 13 27 14
Sum is  54
Average is  18
Product is  4914
Smallest is  13
Largest is  27
```

۲-۲۰ برنامه‌ای بنویسید که از طریق کاربر مقدار شعاع یک دایره را دریافت و میزان قطر دایره، محیط و مساحت آنرا نمایش درآورد. از فرمول‌های زیر استفاده کنید (r شعاع دایره است):

قطر=2r، محیط دایره= $2\pi r$، مساحت= πr^2. $\pi = 3.14159$

۲-۲۱ برنامه‌ای بنویسید که بتواند یک مستطیل، یک لوزی، یک فلش و یک لوزی را همانند الگوهای زیر با استفاده از کاراکتر * ترسیم کند:

```
*********          ***          *            *
*       *        *   *        ***          * *
*       *        *   *       *****        *   *
*       *        *   *         *         *     *
*       *        *   *         *        *       *
*       *        *   *         *         *     *
*       *        *   *         *          * *
*       *        *   *         *           *
*********          ***         *
```

۲-۲۲ کد زیر چه عبارتی چاپ می‌کند؟

```
cout << "*\n**\n***\n****\n*****\n"<<endl;
```

۲-۲۳ برنامه‌ای بنویسید که پنج عدد در یافت و بزرگترین و کوچکترین آنها را چاپ کند. از تکنیک‌های معرفی شده در این فصل استفاده کنید.

۲-۲٤ برنامه‌ای بنویسید که یک عدد صحیح دریافت و تعیین نماید که آیا آن عدد زوج است یا فرد؟

۲-۲۵ برنامه‌ای بنویسید که دو عدد صحیح دریافت و تعیین نماید که آیا عدد اولی حاصلضربی از عدد دوم است یا خیر؟

۲-۲٦ با استفاده از هشت عبارت خروجی، الگوی زیر را تولید و سپس همین الگو را در صورت امکان با استفاده از چهار عبارت خروجی پیاده‌سازی نمائید.

```
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
```

۲-۲۷ در این فصل با اعداد صحیح و نوع **int** آشنا شدید. با این همه C++ قادر به نمایش حروف کوچک، بزرگ و انواع متنوعی از نمادهای خاص نیز است. زبان C++ با استفاده از مقادیر داخلی صحیح و کوچک مبادرت به نمایش کاراکترها می‌کند. مجموعه کاراکترهای بکار رفته توسط یک کامپیوتر و مقادیر صحیح متناظر با آن کاراکترها، مجموعه کاراکتری کامپیوتر نامیده می‌شود. می‌توان یک کاراکتر را با قرار دادن آن کاراکتر مابین یک گوتیشن (*single quotes*) چاپ کرد. برای مثال:

```
cout << 'A'; // print an uppercase A
```

می‌توان معادل صحیح یک کاراکتر را با استفاده از ***static_cast*** و بفرم زیر چاپ کرد:

```
cout <<static_cast< int >( 'A' ); //print 'A' as an integer
```

که به اینحالت عملیات ***cast*** گفته می‌شود (در فصل چهارم به معرفی جامعتر cast خواهیم پرداخت). هنگامی که عبارت فوق اجرا گردد، مقدار 65 را چاپ خواهد کرد (بر روی سیستمی که از مجموعه کاراکتری ASCII استفاده می‌کند).

برنامه‌ای بنویسد که معادل صحیح تعدادی از حروف بزرگ، کوچک، ارقام و نمادهای ویژه را چاپ کند.

۲-۲۸ برنامه‌ای بنویسید که یک عدد متشکل از پنج رقم از کاربر دریافت کرده و سپس آن عدد را به ارقام مجزا از هم تبدیل و هر یک را با سه فاصله از رقم بعدی به چاپ رساند. برای مثال اگر کاربر عدد 42339 را وارد کند، خروجی برنامه باید:

```
4 2 3 3 9
```

باشد. از پنجره دستور برای دریافت ورودی و نمایش خروجی استفاده کنید [این تمرین با استفاده از تکنیک‌های گفته شده در این فصل قابل اجرا است. نیاز است تا از عملیات تقسیم و باقیمانده برای جدا کردن هر رقم استفاده کنید].

۲-۲۹ فقط با استفاده از تکنیک‌های برنامه‌نویسی معرفی شده در این فصل، برنامه‌ای بنویسید که مربع و مکعب اعداد از 0 تا 5 را محاسبه و نتایج آنرا در جدولی بصورت زیر به نمایش درآورد:

integer	square	cube
0	0	0
1	1	1
2	4	8
3	9	27
4	16	64
5	25	125

برای نمایش و دریافت ورودی از پنجره دستور استفاده کنید [نکته: این برنامه نیازی به دریافت ورودی از سوی کاربر ندارد.]

فصل سوم

مقدمه‌ای بر کلاس‌ها و شی‌ها

اهداف

- کلاس‌ها، شی‌ها، توابع عضو و داده چیستند.
- نحوه تعریف کلاس و استفاده از آن در ایجاد کلاس.
- نحوه تعریف توابع عضو در کلاس برای پیاده‌سازی رفتار کلاس.
- نحوه اعلان اعضای داده در کلاس برای پیاده‌سازی صفات کلاس.
- فراخوانی تابع عضو یک شی برای اینکه تابع عضو وظیفه خود را به انجام رساند.
- تفاوت موجود مابین اعضای داده یک کلاس و متغیرهای محلی یک تابع.
- نحوه استفاده از سازنده برای اطمینان از اینکه داده یک شی به هنگام ایجاد شی مقداردهی اولیه شده است.
- نحوه طراحی یک کلاس برای مجزا شدن واسط آن از بخش پیاده‌سازی و افزودن قابلیت استفاده مجدد به آن.

رئوس مطالب

۳-۱ مقدمه

در فصل دوم، چند برنامه ساده ایجاد کردیم که می‌توانستند پیغام‌های را به کاربر نشان داده، اطلاعاتی از وی دریافت نمایند، محاسباتی انجام داده و تصمیم‌گیری کنند. در این فصل، شروع به نوشتن برنامه‌هایی می‌کنیم که مفاهیم پایه برنامه‌نویسی شی‌گرای معرفی شده در بخش ۱۷-۱ را بکار می‌گیرند. یکی از ویژگی‌های مشترک در هر برنامه فصل دوم این است که تمام عبارات که کاری انجام می‌دهند در تابع **main** جای داده شده‌اند. بطور کلی، برنامه‌هایی که در این کتاب ایجاد می‌کنید، متشکل از تابع **main** و یک یا چند کلاس که هر یک حاوی اعضای داده و توابع عضو هستند، خواهند بود. اگر شما عضوی از تیم توسعه (طراحی) در یک مجموعه حرفه‌ای هستید، احتمال دارد بر روی سیستم‌های نرم‌افزاری که حاوی صدها، یا هزاران کلاس است، کار کنید. در این فصل، هدف ما توسعه یک چهارچوب کاری ساده خوش‌فرم به لحاظ مهندسی نرم‌افزار به منظور سازماندهی برنامه‌های شی‌گرا در C++ است.

ابتدا توجه خود را به سمت کلاس‌های موجود در دنیای واقعی متمرکز می‌کنیم. سپس به آرامی به سمت دنباله‌ای از هفت برنامه کامل می‌رویم که به توصیف نحوه ایجاد و استفاده از کلاس‌های متعلق به خودمان می‌پردازند. این مثال‌ها با مبحث آموزشی هدفمند که بر توسعه کلاس *grade-book* تمرکز دارد، شروع می‌شود و مربی می‌تواند از آن برای نگهداری امتیازات دانشجو استفاده کند. این مبحث آموزشی در چند فصل آتی بهبود خواهد یافت و در فصل هفتم بحد اعلی خود خواهد رسید.

۳-۲ کلاس‌ها، شی‌ها، توابع عضو و داده

اجازه دهید بحث را با یک مقایسه ساده که در افزایش درک شما از مطالب بخش ۱۷-۱ موثر است آغاز کنیم. فرض کنید که می‌خواهید با اتومبیلی رانندگی کرده و با فشردن پدال گاز آن را سریعتر به حرکت در آورید. چه اتفاقی قبل از اینکه بتوانید اینکار را انجام دهید، باید رخ دهد؟ بسیار خوب، قبل از اینکه بتوانید با اتومبیلی رانندگی کنید، باید کسی آن را طراحی و ساخته باشد. معمولاً ساخت اتومبیل، با ترسیم یا نقشه‌کشی مهندسی شروع شود. همانند طراحی صورت گرفته برای خانه. این ترسیمات شامل طراحی پدال گاز است که راننده با استفاده از آن سبب می‌شود تا اتومبیل سریعتر حرکت کند. تا حدی، پدال سبب «پنهان» شدن پیچیدگی مکانیزمی می‌شود که اتومبیل را سریعتر بحرکت در می‌آورد، همانطوری که پدال ترمز سبب «پنهان» شدن مکانیزمی می‌شود که از سرعت اتومبیل کم می‌کند، فرمان اتومبیل سبب «پنهان» شدن مکانیزمی می‌شود که اتومبیل را هدایت می‌کند و موارد دیگر. با انجام چنین کارهایی، افراد عادی می‌توانند به آسانی اتومبیل را هدایت کرده و براحتی از پدال گاز، ترمز و فرمان، مکانیزم تعویض دنده و سایر «واسط‌های» کاربرپسند و ساده استفاده کنند تا پیچیدگی مکانیزم‌های داخلی اتومبیل برای راننده مشخص نباشد.

متاسفانه، نمی‌توانید با نقشه‌های ترسیمی یک اتومبیل رانندگی کنید، قبل از اینکه با اتومبیلی رانندگی کنید باید، آن اتومبیل از روی نقشه‌های ترسیمی ساخته شود. یک اتومبیل کاملاً ساخته شده دارای پدال گاز واقعی برای به حرکت درآوردن سریع اتومبیل است. اما این هم کافی نیست، اتومبیل بخودی خود شتاب نمی‌گیرد، از اینرو لازم است راننده بر روی پدال گاز فشار آورد تا به اتومبیل دستور حرکت سریع‌تر را صادر کند.

حال اجازه دهید تا از مثال اتومبیل مطرح شده برای معرفی مفاهیم کلیدی برنامه‌نویسی شی‌گرا در این بخش استفاده کنیم. انجام یک وظیفه در یک برنامه مستلزم یک تابع (همانند **main** که فصل دوم توضیح داده شده) است. تابع، توصیف‌کننده مکانیزمی است که وظیفه واقعی خود را به انجام می‌رساند. تابع پیچیدگی وظایفی که قرار است انجام دهد از دید کاربر خود پنهان می‌سازد، همانند پدال گاز در اتومبیل که پیچیدگی مکانیزم شتاب‌گیری را از دید راننده پنهان می‌کند. در C++، کار را با ایجاد واحدی بنام کلاس که خانه تابع محسوب می‌شود، آغاز می‌کنیم، همانند نقشه ترسیمی اتومبیل که طرح پدال گاز نیز در آن قرار دارد. از بخش ۱۷-۱ بخاطر دارید که به تابع متعلق به یک کلاس، تابع عضو گفته می‌شود. در یک کلاس می‌توان یک یا چند تابع عضو داشت که برای انجام وظایف کلاس طراحی شده‌اند. برای مثال، یک کلاس می‌تواند نشان‌دهنده یک حساب بانکی و حاوی یک تابع عضو برای میزان سپرده در

حساب، تابع دیگری برای میزان برداشت پول از حساب و تابع سومی هم برای نمایش میزان پول موجود در حساب باشد.

همانطوری که نمی‌توانید با نقشه ترسیمی اتومبیل رانندگی کنید، نمی‌توانید کلاسی را مشتق کنید. همانطوری که باید قبل از اینکه بتوانید با اتومبیلی رانندگی کنید، شخص از روی نقشه ترسیمی مبادرت به ساخت اتومبیل کرده باشد، شما هم باید قبل از اینکه برنامه بتواند وظایف توصیفی در کلاس را بدست آورد، باید یک شی از کلاس را ایجاد کرده باشید. این یکی از دلایل شناخته شدن C++ بعنوان یک زبان برنامه‌نویسی شی‌گرا است. همچنین توجه نمائید که همانطوری که می‌توان از روی نقشه ترسیمی اتومبیل‌های متعددی ساخت، می‌توان از روی یک کلاس، شیی‌های متعددی ایجاد کرد.

زمانیکه رانندگی می‌کنید، با فشردن پدال گاز، پیغامی به اتومبیل ارسال می‌شود که وظیفه‌ای را انجام دهد، که این وظیفه افزودن سرعت اتومبیل است. به همین ترتیب، پیغام‌هایی را به یک شی ارسال می‌کنیم، هر پیغام بعنوان فراخوان یک تابع عضو شناخته می‌شود و به تابع عضو از شیی اعلان می‌کند که وظیفه خود را انجام دهد. اینکار غالباً بعنوان تقاضای سرویس از یک شی شناخته می‌شود.

تا بدین‌جا، از مقایسه اتومبیل برای توضیح کلاس‌ها، شی‌ها و توابع عضو استفاده کردیم. علاوه بر قابلیت‌های اتومبیل، هر اتومبیلی دارای چندین صفت است، صفاتی همانند رنگ، تعداد درها، ظرفیت باک، سرعت جاری و مسافت طی شده. همانند قابلیت‌های اتومبیل، این صفات هم بعنوان بخشی از طراحی اتومبیل در نقشه ترسیمی دیده می‌شوند. همانطوری که رانندگی می‌کنید، این صفات همیشه در ارتباط با اتومبیل هستند. هر اتومبیلی، حافظ صفات خود است. برای مثال، هر اتومبیلی از میزان سوخت موجود در باک خود مطلع است، اما از میزان سوخت موجود در باک سایر اتومبیل‌ها مطلع نیست. به همین ترتیب، یک شی دارای صفاتی است که به همراه شی بوده و در برنامه بکار گرفته می‌شوند.

این صفات به عنوان بخشی از کلاس شی تصریح می‌شوند. برای مثال، یک شی حساب بانکی دارای صفت موجودی است که نشاندهنده مقدار پول موجود در حساب می‌باشد. هر شی حساب بانکی از میزان موجودی در حساب خود مطلع است، اما از موجودی سایر حساب‌ها در بانک اطلاعی ندارد. صفات توسط اعضای داده کلاس مشخص می‌شوند.

۳-۳ نگاهی بر مثال‌های این فصل

در مابقی این فصل مبادرت به معرفی هفت مثال ساده می‌کنیم که به توصیف مفاهیم معرفی شده در ارتباط با قیاس اتومبیل می‌پردازند. خلاصه‌ای از این مثال‌ها در زیر آورده شده است. ابتدا مبادرت به ایجاد کلاس **GradeBook** می‌کنیم تا قادر به توصیف این مفاهیم باشیم:

۱- اولین مثال نشاندهنده کلاس **GradeBook** با یک تابع عضو است که فقط یک پیغام خوش‌آمدگویی را در زمان فراخوانی به نمایش در می‌آورد. سپس شما را با نحوه ایجاد یک شی از این کلاس و فراخوانی تابع عضو که پیغام خوش‌آمدگویی را ظاهر می‌سازد، آشنا خواهیم کرد.

۲- دومین مثال، اصلاح شده مثال اول است. در این مثال به تابع عضو اجازه داده می‌شود تا نام دوره را به عنوان یک آرگومان قبول کند. سپس، تابع عضو مبادرت به نمایش نام دوره بعنوان بخشی از پیغام خوش‌آمدگویی می‌کند.

۳- سومین مثال نحوه ذخیره‌سازی نام دوره در یک شی **GradeBook** را نشان می‌دهد. برای این نسخه از کلاس، شما را با نحوه استفاده از توابع عضو به منظور تنظیم نام دوره در شی و بدست آوردن نام دوره از شی آشنا می‌کنیم.

۴- چهارمین مثال به توصیف نحوه مقداردهی اولیه داده در شی **GradeBook** به هنگام ایجاد شی می‌پردازد. مقداردهی اولیه توسط یک تابع عضو بنام سازنده کلاس صورت می‌گیرد. همچنین این مثال نشان می‌دهد که هر شی **GradeBook** از نام دورهٔ خود نگهداری می‌کند.

۵- مثال پنجم اصلاح شده مثال چهارم است که به توصیف نحوه قرار دادن کلاس **GradeBook** در یک فایل مجزا به منظوراستفاده مجدد از نرم‌افزار، می‌پردازد.

۶- مثال ششم، تغییریافته مثال پنجم است که حاوی یک اصل مهندسی نرم‌افزار در جداسازی بخش واسط کلاس از بخش پیاده‌سازی آن است. با انجام اینکار فرآیند اصلاح و تغییر آسانتر می‌شود بدون اینکه در سرویس‌گیرنده‌های کلاس تاثیرگذار باشد.

۷- آخرین مثال قابلیت کلاس **GradeBook** را با معرفی اعتبارسنجی داده افزایش خواهد داد. اعتبارسنجی ما را از فرمت صحیح داده در یک شی مطمئن می‌سازد. برای مثال یک شی **Date** نیازمند یک مقدار در محدودهٔ 1 الی 12 است. در این مثال **GradeBook**، تابع عضوی که مبادرت به تنظیم نام دورهٔ برای یک شی **GradeBook** می‌کند، سبب می‌شود تا نام دوره 25 کاراکتر یا کمتر باشد. اگر طول

نام دوره بیش از 25 کاراکتر باشد، تابع عضو فقط از 25 کاراکتر اول استفاده کرده و یک پیغام خطا به نمایش در می‌آورد.

توجه کنید که مثال‌های **GradeBook** در این فصل فرآیند واقعی ذخیره‌سازی را انجام نمی‌دهند. در فصل چهارم، فرآیند پردازش امتیازات را با **GradeBook** و در فصل هفتم مبادرت به ذخیره‌سازی اطلاعات در یک شی **GradeBook** خواهیم کرد.

۳-٤ تعریف کلاس با یک تابع عضو

کار را با مثالی آغاز می‌کنیم (شکل ۱-۳) که حاوی کلاس **GradeBook** است و می‌توان از آن برای نگهداری امتیازات دانشجویان استفاده کرد. تابع **main** در خطوط 25-20 مبادرت به ایجاد یک شی **GradeBook** کرده است. این برنامه نسخه اول بوده و نسخه کامل آن در فصل هفتم ایجاد خواهد شد. تابع **main** از این شی و تابع عضو آن برای نمایش پیغام بر روی صفحه نمایش (پیغام خوش‌آمدگویی) استفاده می‌کند.

```
1  // Fig. 3.1: fig03_01.cpp
2  // Define class GradeBook with a member function displayMessage;
3  // Create a GradeBook object and call its displayMessage function.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  // GradeBook class definition
9  class GradeBook
10 {
11 public:
12    // function that displays a welcome message to the GradeBook user
13    void displayMessage()
14    {
15       cout << "Welcome to the Grade Book!" << endl;
16    } // end function displayMessage
17 }; // end class GradeBook
18
19 // function main begins program execution
20 int main()
21 {
22    GradeBook myGradeBook; // create a GradeBook object named myGradeBook
23    myGradeBook.displayMessage(); // call object's displayMessage function
24    return 0; // indicate successful termination
25 } // end main
```

```
Welcome to the Grade Book!
```

شکل ۱-۳ | تعریف کلاس GradeBook با یک تابع عضو، ایجاد یک شی GradeBook و فراخوانی تابع عضو آن.

ابتدا به توصیف نحوه تعریف یک کلاس و تابع عضو می‌پردازیم. سپس به توضیح نحوه ایجاد یک شی و نحوه فراخوانی تابع عضو یک شی می‌پردازیم. چند مثال اول حاوی تابع **main** و کلاس **GradeBook** است که از آن در همان فایل استفاده می‌کند. در انتهای این فصل، به معرفی روش‌های

خبره در ساختارمند کردن برنامه‌ها به منظور افزایش کارایی و انجام بهتر مهندسی نرم‌افزار خواهیم پرداخت.

کلاس GradeBook

قبل از اینکه تابع **main** (خطوط 25-20) بتواند شی از کلاس **GradeBook** ایجاد کند، بایستی به کامپایلر توابع عضو و اعضای داده متعلق به کلاس را اعلان کنیم. این کار بنام تعریف کلاس شناخته می‌شود. تعریف کلاس **GradeBook** (خطوط 17-9) حاوی تابع عضوی بنام **displayMessage** است (خطوط 16-13) که پیغامی بر روی صفحه نمایش چاپ می‌کند (خط 15). بخاطر دارید که یک کلاس همانند یک نقشه ترسیمی است، از اینرو نیاز داریم که یک شی از کلاس **GradeBook** ایجاد کنیم (خط 22)، و تابع عضو **displayMessage** آن را فراخوانی نمایم (خط 23) تا خط 15 را به اجرا درآورده و پیغام خوش‌آمدگویی را به نمایش درآورد. بزودی به توضیح دقیق‌تر خطوط 23-22 خواهیم پرداخت.

تعریف کلاس از خط 9 و با کلمه کلیدی ***class*** و بدنبال آن نام کلاس **GradeBook** آغاز شده است. بطور قراردادی، نام کلاس‌های تعریف شده توسط کاربر با حروف بزرگ شروع می‌شوند و در صورت وجود کلمات بعدی، ابتدای آنها هم با حروف بزرگ آغاز می‌گردد. به این روش غالباً روش *camel case* می‌گویند.

بدنه هر کلاسی در بین جفت براکت باز و بسته (**{** و **}**) محدود می‌شود، همانند خطوط 10 و17. تعریف کلاس با یک سیمکولن خاتمه می‌پذیرد (خط 17).

خطای برنامه‌نویسی

فراموش کردن سیمکولن در انتهای تعریف یک کلاس، خطای نحوی بدنبال خواهد داشت.

بخاطر دارید که تابع **main** همیشه و بصورت اتوماتیک به هنگام اجرای برنامه فراخوانی می‌شود. اکثر توابع بصورت اتوماتیک فراخوانی نمی‌شوند. همانطوری که مشاهده خواهید کرد، بایستی تابع عضو **displayMessage** را بطور صریح فراخوانی کنید تا وظیفه خود را انجام دهد.

خط 14 حاوی برچسب تصریح‌کننده دسترسی **public:** است. کلمه کلیدی **public**، تصریح‌کننده دسترسی نامیده می‌شود. خطوط 16-13 تعریف‌کننده تابع عضو **displayMessage** هستند. این تابع عضو پس از **public:** قرار دارد تا نشان دهد که تابع «بصورت سراسری در دسترس است»، به این معنی که می‌تواند توسط سایر توابع موجود در برنامه و توسط توابع عضو کلاس‌های دیگر فراخوانی گردد. همیشه در مقابل تصریح‌کننده‌های دسترسی یک کولن (:) قرار داده می‌شود. برای مابقی متن، زمانیکه به

تصریح‌کننده دسترسی **public** اشاره می‌کنیم، کولن آن را در نظر نمی‌گیریم. در بخش ۶-۳ به معرفی دومین تصریح‌کننده دسترسی بنام ***private*** خواهیم پرداخت.

هر تابعی در برنامه وظیفه‌ای را انجام داده و امکان دارد مقداری را پس از کامل کردن وظیفه خود برگشت دهد. برای مثال، تابعی می‌تواند یک محاسبه انجام داده و سپس نتیجه آن محاسبه را برگشت دهد. زمانیکه تابعی تعریف می‌شود، باید نوع برگشتی آن را مشخص سازید، تا تابع پس از اتمام کار، آن نوع را برگشت دهد. در خط 13، کلمه کلیدی **void** قرار گرفته است (در سمت چپ نام تابع **displayMessage**) و نشاندهندهٔ نوع برگشتی از سوی تابع است. نوع برگشتی **void** بر این نکته دلالت دارد که **displayMessage** وظیفه‌ای را انجام خواهد داد، اما داده‌ای را به فراخوان خود (در این مثال، تابع فراخوان، **main** می‌باشد) پس از اتمام وظیفه خود برگشت نخواهد داد. (در شکل ۵-۳ مثالی را مشاهده می‌کنید که تابع در آن مقداری را برگشت داده است).

نام تابع عضو **displayMessage** بوده و قبل از آن نوع برگشتی قرار دارد. بطور قراردادی، اسامی توابع با یک حرف کوچک شروع شده و تمام کلمات متعاقب آن با حرف بزرگ شروع می‌شوند. پرانتزهای قرار گرفته پس از نام تابع عضو، بر این نکته دلالت دارند که این یک تابع است. یک جفت پرانتز خالی، همانند خط 13، نشان می‌دهد که این تابع عضو، نیازی به داده اضافی برای انجام وظیفه خود ندارد. در بخش ۵-۳ شاهد تابع عضوی خواهید بود که برای انجام وظیفه خود نیازمند داده اضافی است. معمولاً به خط 13، *سرآیند تابع* گفته می‌شود. بدنه هر تابعی با براکت‌های باز و بسته مشخص می‌شود ({ و })، همانند خطوط 14 و 16.

بدنه تابع حاوی عباراتی است که وظیفه تابع را به انجام می‌رسانند. در این مورد، تابع عضو displayMessage حاوی یک عبارت است (خط 15) که پیغام "Welcome to the Grade Book!" را به نمایش در می‌آورد. پس از اجرای این عبارت، تابع وظیفه خود را به انجام رسانده است.

خطای برنامه‌نویسی

*برگشت دادن مقداری از یک تابع که نوع برگشتی آن بصورت **void** اعلان شده است، خطای کامپایلر بدنبال خواهد داشت.*

خطای برنامه‌نویسی

تعریف تابعی در درون تابع دیگر، خطای نحوی است.

تست کلاس GradeBook

در این مرحله می‌خواهیم که از کلاس **GradeBook** در برنامه استفاده کنیم. همانطوری که در فصل دوم آموختید، تابع **main** با اجرای هر برنامه‌ای شروع بکار می‌کند. خطوط 25-20 از برنامه شکل ۱–۳ حاوی تابع **main** هستند، که اجرای برنامه را تحت کنترل دارد.

در این برنامه، مایل هستیم تابع عضو **displayMessage** از کلاس **GradeBook** برای نمایش پیغام خوش‌آمدگویی اجرا گردد (فراخوانی شود). اصولاً تا زمانیکه شیی از یک کلاس ایجاد نکرده باشید، نمی‌توانید تابع عضو کلاس را فراخوانی نمائید (در بخش ۷–۱۰ با تابع عضو **static**، آشنا خواهید شد که در این مورد یک استثناء می‌باشد.) خط 22 یک شی از کلاس **GradeBook** بنام **myGradeBook** ایجاد می‌کند. دقت کنید که نوع متغیر **GradeBook** است، کلاس تعریف شده در خطوط 17-9. زمانیکه متغیرهایی از نوع **int** اعلان می‌کنیم، همانند کاری که در فصل دوم انجام دادیم، کامپایلر از مفهوم **int** مطلع است و می‌داند که آن یک نوع بنیادین می‌باشد. با این وجود، زمانیکه مبادرت به نوشتن خط 22 می‌کنیم، کامپایلر بصورت اتوماتیک از مفهوم نوع **GradeBook** اطلاعی ندارد و آن را نمی‌شناسد، این نوع یک نوع تعریف شده توسط کاربر است. از اینرو، باید به کامپایلر، با قرار دادن تعریف کلاس همانند کاری که در خطوط 17-9 انجام داده‌ایم، **GradeBook** را معرفی کنیم. اگر این خطوط را حذف کنیم، کامپایلر پیغام خطایی همانند "*'GradeBook' : undeclared identifier* " را در GNU C++ صادر می‌کند. هر کلاس جدیدی که ایجاد می‌کنید، تبدیل به یک نوع جدید می‌شود که می‌تواند برای ایجاد شی‌ها بکار گرفته شود. برنامه‌نویسان می‌توانند در صورت نیاز مبادرت به تعریف کلاس‌های با نوع جدید نمایند، و این یکی از دلایل شناخته شدن C++ بعنوان یک زبان بسط‌پذیر است.

خط 23 مبادرت به فراخوانی تابع عضو **displayMessage** (تعریف شده در خطوط 16-13) با استفاده از متغیر **myGradeBook** و بدنبال آن عملگر نقطه (.)، سپس نام تابع **displayMessage** و یک جفت پرانتز خالی می‌کند. این فراخوانی موجب می‌شود که تابع **displayMessage** برای انجام وظیفه خود فراخوانی شود. در ابتدای خط 23، عبارت "**myGradeBook.**" بر این نکته دلالت دارد که **main** بایستی از شی **GradeBook** که در خط 22 ایجاد شده است، استفاده نماید. پرانتزهای خالی در خط 13 نشان می‌دهند که تابع عضو **displayMessage** نیازی به داده اضافی برای انجام وظیفه خود ندارد. (در بخش ۵–۳، با نحوه ارسال داده به تابع آشنا خواهید شد) زمانیکه **displayMessage** وظیفه خود را انجام داد، تابع **main** به اجرای خود از خط 24 ادامه خواهد داد، که نشان می‌دهد **main** وظیفه خود را به بدرستی انجام داده است. با رسیدن به انتهای **main**، برنامه خاتمه می‌پذیرد.

دیاگرام کلاس UML برای کلاس GradeBook

از بخش ۱۷-۱ بخاطر دارید که UML یک زبان گرافیکی بکار رفته توسط برنامه‌نویس برای نمایش سیستم‌های شی‌گرا به یک روش استاندارد است. در UML، هر کلاس در یک دیاگرام کلاس و بصورت یک مستطیل با سه قسمت (بخش) مدل‌سازی می‌شود. شکل ۲-۳ نشاندهنده یک دیاگرام کلاس UML برای کلاس **GradeBook** معرفی شده در برنامه ۱-۳ است. بخش فوقانی حاوی نام کلاس است، که در وسط قرار گرفته و بصورت توپر نوشته می‌شود. بخش میانی، حاوی صفات کلاس است که مرتبط با اعضای داده در C++ است. در شکل ۲-۳ بخش میانی خالی است، چرا که این نسخه از کلاس **GradeBook** در برنامه ۱-۳ دارای صفات نیست. (در بخش ۶-۳ نسخه‌ای از کلاس **GradeBook** عرضه شده که دارای یک صفت است.) بخش تحتانی حاوی عملیات کلاس است، که متناظر با توابع عضو در C++ می‌باشد.

شکل ۲-۳ | دیاگرام کلاس UML نشان می‌دهد که کلاس GradeBook دارای یک عملیات سراسری بنام displayMessage است.

UML مبادرت به مدلسازی عملیات‌ها با لیست کردن نام عملیات و بدنبال آن مجموعه پرانتزها می‌کند. کلاس **GradeBook** فقط دارای یک تابع عضو بنام **displayMessage** است، از اینرو بخش تحتانی در شکل ۲-۳ فقط یک عملیات با این نام را لیست کرده است. تابع عضو **displayMessage** برای انجام وظیفه خود نیازی به اطلاعات اضافی ندارد، از اینرو پرانتزهای قرار گرفته پس از **displayMessage** در این دیاگرام کلاس خالی هستند، همانند سرآیند تابع عضو قرار گرفته در خط 13 برنامه۱-۳. علامت جمع (+) که قبل از نام عملیات آورده شده، نشان می‌دهد که **displayMessage** یک عملیات سراسری در UML است (یک تابع عضو **public** در C++). به دفعات از دیاگرام‌های کلاس UML برای خلاصه کردن صفات و عملیات کلاس‌ها استفاده خواهیم کرد.

۳-۵ تعریف تابع عضو با پارامتر

در مثال قیاس اتومبیل در بخش ۲-۳ خاطرنشان کردیم که فشردن پدال گاز سبب ارسال پیغامی به اتومبیل می‌شود تا وظیفه‌ای را به انجام برساند، که در این مورد افزودن سرعت اتومبیل است. اما اتومبیل باید چقدر سرعت بگیرد؟ همانطوری که می‌دانید، با فشردن هر چه بیشتر پدال، سرعت اتومبیل افزایش پیدا می‌کند. بنابر این پیغام ارسالی به اتومبیل هم حاوی وظیفه بوده و همچنین حاوی اطلاعات دیگری است که به اتومبیل در انجام وظیفه کمک می‌کند. این اطلاعات اضافی بنام پارامتر شناخته می‌شوند، مقدار پارامتر به اتومبیل کمک می‌کند تا میزان افزایش سرعت را تعیین کند. به همین ترتیب، یک تابع عضو می‌تواند نیازمند یک یا چندین پارامتر بعنوان داده اضافی برای انجام وظیفه خود باشد. فراخوان تابع مقادیری بنام آرگومان، برای هر پارامتر تابع تدارک می‌بیند. برای مثال، در سپرده‌گذاری در حساب

بانکی، فرض کنید تابع عضو **deposit** از کلاس **Account** پارامتری که نشاندهنده مقدار سپرده است، مشخص کرده باشد. زمانیکه تابع عضو **deposit** فراخوانی شود، مقدار آرگومان که نشاندهنده مقدار سپرده است به پارامتر تابع عضو کپی می‌شود. سپس تابع عضو این مقدار سپرده را به میزان موجودی اضافه می‌کند.

تعریف و تست کلاس GradeBook

مثال بعدی (برنامه شکل ۳-۳) تعریف مجددی از کلاس **GradeBook** (خطوط 23-14) با تابع عضو **displayMessage** است (خطوط 22-18) که نام دوره را بعنوان بخشی از رشته خوش‌آمدگویی چاپ می‌کند. تابع عضو جدید **displayMessage** مستلزم یک پارامتر است (**courseName** در خط 18) که نشاندهنده نام دوره است.

قبل از اینکه به بررسی ویژگی‌های جدید کلاس **GradeBook** بپردازیم، اجازه دهید به نحوه استفاده از کلاس جدید در **main** نگاهی داشته باشیم (خطوط 40-26). در خط 28 یک متغیر از نوع رشته (***string***) بنام **nameOfCourse** ایجاد شده است که از آن برای ذخیره کردن نام دوره وارد شده توسط کاربر استفاده خواهیم کرد. متغیر از نوع **string** نشاندهنده رشته‌ای از کاراکترها همانند "CS101 Introduction to C++ Programming" است. نوع **string** در واقع شی از کلاس **string** کتابخانه استاندارد C++ است. این کلاس در فایل سرآیند <**string**> تعریف شده و نام **string** همانند **cout**، متعلق به فضای‌نامی **std** است. برای اینکه خط 28 قادر به کامپایل شدن باشد، خط 9 حاوی سرآیند <**string**> است. دقت کنید که اعلان **using** در خط 10 به ما اجازه می‌دهد تا به آسانی در خط 28 از **string** بجای **std::string** استفاده کنیم. برای این لحظه، می‌توانید در مورد متغیرهای **string** همانند متغیرهای از نوع دیگر همانند **int** فکر کنید. در بخش ۱۰-۳ با قابلیت‌های **string** بیشتر آشنا خواهید شد.

خط 29 یک شی از کلاس **GradeBook** بنام **myGradeBook** ایجاد می‌کند. خط 32 به کاربر اعلان می‌کند که نام دوره را وارد سازد. خط 33 مبادرت به خواندن نام دوره از ورودی کاربر کرده و آن را با استفاده از تابع کتابخانه‌ای **getline** به متغیر **nameOfCourse** تخصیص می‌دهد. قبل از اینکه به توضیح این خط از کد بپردازیم، اجازه دهید تا توضیح دهیم چرا نمی‌توانیم برای بدست آوردن نام دوره فقط بنویسیم

```
cin>>nameOfCourse;
```

در اجرای نمونه این برنامه، ما از نام دوره "CS101 Introduction to C++ Programming" استفاده کرده‌ایم که حاوی چندین کلمه است. زمانیکه **cin** با عملگر استخراج به کار گرفته می‌شود، مبادرت به خواندن کاراکترها تا رسیدن به اولین کاراکتر *white-space* می‌کند. از اینرو فقط "CS101" توسط این عبارت خوانده می‌شود و برای خواندن مابقی نام دوره مجبور به عملیات ورودی اضافی خواهیم بود.

در این مثال، مایل هستیم تا کاربر نام کامل دوره را تایپ کرده و کلید *Enter* را برای تحویل آن به برنامه فشار دهد، و کل نام دوره در متغیر رشته‌ای **nameOfCourse** ذخیره گردد فراخوانی تابع **getline(**cin, nameOfCourse**)** در خط 33، سبب می‌شود که کاراکترها (از جمله کاراکترهای فاصله که کلمات را از هم در ورودی جدا می‌کنند) از شی ورودی استاندارد جریان **cin** (صفحه کلید) تا رسیدن به کاراکتر خط جدید خوانده شده و در متغیر رشته‌ای **nameOfCourse** جای داده شده و کاراکتر خط جدید حذف می‌گردد. توجه نمائید زمانیکه کلید *Enter* فشرده می‌شود (در زمان تایپ ورودی)، یک خط جدید (*newline*) به جریان ورودی افزوده می‌شود. همچنین توجه کنید که فایل سرآیند <**string**> باید قرار داشته باشد تا بتوان از تابع **getline** استفاده کرد. نام این تابع متعلق به فضای نامی **std** است.

```
1  // Fig. 3.3: fig03_03.cpp
2  // Define class GradeBook with a member function that takes a parameter;
3  // Create a GradeBook object and call its displayMessage function.
4  #include <iostream>
5  using std::cout;
6  using std::cin;
7  using std::endl;
8
9  #include <string> // program uses C++ standard string class
10 using std::string;
11 using std::getline;
12
13 // GradeBook class definition
14 class GradeBook
15 {
16 public:
17    // function that displays a welcome message to the GradeBook user
18    void displayMessage( string courseName )
19    {
20       cout << "Welcome to the grade book for\n" << courseName << "!"
21          << endl;
22    } // end function displayMessage
23 }; // end class GradeBook
24
25 // function main begins program execution
26 int main()
27 {
28    string nameOfCourse; // string of characters to store the course name
29    GradeBook myGradeBook; // create a GradeBook object named myGradeBook
30
31    // prompt for and input course name
32    cout << "Please enter the course name:" << endl;
```

```
33    getline( cin, nameOfCourse ); // read a course name with blanks
34    cout << endl; // output a blank line
35
36    // call myGradeBook's displayMessage function
37    // and pass nameOfCourse as an argument
38    myGradeBook.displayMessage( nameOfCourse );
39    return 0; // indicate successful termination
40 } // end main
```

```
Please enter the course name:
CS101 Introduction to C++ Programming

Welcome to the grade book for
CS101 Introduction to C++ Programming!
```

شکل ۳-۳ | تعریف کلاس GradeBook با تابع عضوی که پارامتر دریافت می‌کند.

خط 38 تابع عضو **displayMessage** شی **myGradeBook** را فراخوانی می‌کند. **متغیر nameOfCource** موجود در درون پرانتزها آرگومانی است که به تابع **displayMessage** ارسال می‌شود، از اینروست که تابع می‌تواند وظیفه خود را به انجام رساند. مقدار متغیر **nameOfCourse** در **main** تبدیل به مقدار پارامتر **courseName** تابع عضو **displayMessage** در خط 18 می‌شود. زمانیکه این برنامه اجرا شود، توجه نمائید که تابع عضو **displayMessage** به همراه پیغام خوش‌آمدگویی، نام دورهٔ تایپ شده توسط کاربر را به نمایش درمی‌آورد.

آرگومان‌ها و پارامترهای بیشتر

برای تصریح اینکه تابعی نی‌ازمند داده برای انجام وظایف خویش است، اطلاعات اضافی را در لیست پارامتری تابع قرار می‌دهیم، لیستی که در درون پرانتزها قرار گرفته پس از نام تابع جای دارد. لیست پارامترها می‌تواند به هر تعداد پارامتر داشته باشد یا اصلاً پارامتری نداشته باشد (همانند پرانتزهای خالی در خط 13 برنامه شکل ۱-۳) تا نشان دهد تابع نیاز به هیچ پارامتری ندارد. لیست پارامتری تابع عضو **displayMessage** به نحوی اعلان شده که تابع نیازمند یک پارامتر است (شکل ۳-۳، خط 18). هر پارامتر باید دارای نوع و یک شناسه باشد. در این مورد، نوع **string** بوده و شناسه **courseName** است و نشان می‌دهد که تابع عضو **displayMessage** مستلزم یک رشته برای انجام وظیفه خویش است. بدنه تابع عضو از پارامتر **courseName** برای دسترسی به مقدار ارسالی به تابع در فراخوانی تابع (خط 38 در **main**) استفاده می‌کند. خطوط 21-20 مقدار پارامتر **courseName** را بعنوان بخشی از پیغام خوش‌آمدگویی به نمایش در می‌آورد. توجه نمائید که نام متغیر پارامتری (خط 18) می‌تواند همنام متغیر آرگومان (خط 38) یا نام متفاوتی داشته باشد. در فصل ششم با دلایل اینکارها آشنا خواهید شد.

یک تابع می‌تواند دارای چندین پارامتر باشد، بشرطی که هر پارامتر از دیگری با یک ویرگول مجزا شده باشد (در مثال شکل‌های ۴-۶ و ۵-۶ شاهد آنها خواهید بود). تعداد و ترتیب آرگومان‌ها در یک تابع فراخوانی شده بایستی با تعداد و ترتیب لیست پارامترها در سرآیند تابع عضو فراخوانی شده مطابقت داشته

باشد. همچنین، نوع آرگومان‌ها در تابع فراخوانی شده باید با نوع پارامترهای متناظر در سرآیند تابع مطابقت داشته باشد. (همانطوری که جلوتر می‌رویم، شاهد خواهید بود که نوع یک آرگومان می‌تواند همانند نوع پارامتر متناظر خود نباشد.) در مثال ما، یک آرگومان رشته‌ای در تابع فراخوانی شده وجود دارد (**nameOfCourse**) که دقیقاً با پارامتری رشته‌ای در تعریف تابع عضو مطابقت دارد (**CourseName**).

خطای برنامه‌نویسی

قراردادن یک سیمکولن پس از پرانتز سمت راست در لیست پارامتری تعریف تابع، خطای نحوی است.

خطای برنامه‌نویسی

تعریف یک پارامتر تابع همانند یک متغیر محلی در تابع، خطای کامپایل بدنبال خواهد داشت.

برنامه‌نویسی ایده‌ال

از ابهام اجتناب کنید، از اسامی یکسان برای آرگومان‌های ارسالی به تابع و پارامترهای متناظر در تعریف تابع، استفاده نکنید.

برنامه‌نویسی ایده‌ال

برای توابع و پارامترها اسامی با معنی انتخاب کنید تا خوانایی برنامه افزایش یافته و نیاز به افزودن توضیحات را کم نماید.

به روز کردن دیاگرام کلاس UML برای کلاس GradeBook

دیاگرام کلاس UML در شکل ۴-۳ مبادرت به مدلسازی کلاس **GradeBook** از برنامه ۳-۳ کرده است. همانند کلاس **GradeBook** تعریف شده در برنامه ۱-۳، این کلاس **GradeBook** حاوی یک تابع عضو سراسری **displayMessage** است. با این وجود، این نسخه از **displayMessage** دارای یک پارامتر می‌باشد. UML پارامتر را با لیست کردن نام پارامتر، بدنبال آن یک کولن و سپس نوع پارامتر در درون پرانتزهای قرار گرفته پس از نام عملیات مدل‌سازی می‌کند. UML دارای نوع داده‌های متعلق بخود شبیه C++ است. UML یک زبان مستقل بوده و از آن در کنار انواع زبان‌های برنامه‌نویسی استفاده می‌شود، از اینرو اصطلاحات تخصصی آن دقیقاً مطابق با C++ نیست. برای مثال، نوع رشته (**String**) در UML متناظر با نوع **string** در C++ است. تابع عضو **displayMessage** از کلاس **GradeBook** (شکل ۳-۳ خطوط 22-18) دارای یک پارامتر رشته‌ای بنام **CourseName** است، از اینرو در شکل ۴-۳ بصورت **courseName : String** در میان پرانتزها و پس از نام **diaplayName**، قرار گرفته است. دقت کنید که این نسخه از کلاس **GradeBook** هنوز دارای اعضای داده نیست.

شکل ٤-٣ | دیاگرام کلاس UML نشان می‌دهد که کلاس GradeBook دارای عملیات displayMessage با یک پارامتر از نوع String در UML است.

٦-٣ اعضای داده، توابع set و get

در فصل دوم، تمام متغیرهای برنامه را در تابع **main** اعلان کردیم. متغیرهای اعلان شده در بدنه تعریف یک تابع، بنام متغیرهای محلی شناخته می‌شوند و می‌توانند فقط از خطی که در آن تابع اعلان شده‌اند تا رسیدن به براکت بستن سمت راست (}) در تعریف تابع بکار گرفته شوند. قبل از اینکه بتوان از یک متغیر محلی استفاده کرد، باید ابتدا آن را اعلان کرده باشید. زمانیکه تابع به کار خود خاتمه می‌دهد، مقادیر متغیرهای محلی آن از بین می‌روند (در فصل ششم با متغیرهای محلی **static** آشنا خواهید شد که از این قاعده مستثنی هستند). از بخش ۲-۳ بخاطر دارید که شیی که دارای صفت است، آن را با خود همراه دارد و در برنامه از آن استفاده می‌شود. چنین صفاتی در کل طول عمر یک شی وجود خواهند داشت.

معمولاً کلاسی که دارای یک یا چندین تابع عضو است، مبادرت به دستکاری کردن صفات متعلق به یک شی مشخص از کلاسی می‌کند. صفات به عنوان متغیرهای در تعریف کلاس عرضه می‌شوند. چنین متغیرهای، بنام اعضای داده شناخته می‌شوند و در درون تعریف کلاس اعلان می‌گردند اما در خارج از تعریف بدنه تابع عضو کلاس جای می‌گیرند. هر شی از کلاس مسئول نگهداری کپی از صفات متعلق به خود در حافظه است. مثال مطرح شده در این بخش به توصیف کلاس **GradeBook** می‌پردازد که حاوی یک عضو داده **CourseName** است تا نشان‌دهنده نام یک دوره خاص از شی **GradeBook** باشد.

کلاس GradeBook با یک عضو داده، یک تابع set و get

در این مثال، کلاس **GradeBook** (برنامه شکل ۵-۳) مبادرت به نگهداری نام دوره بصورت یک عضو داده می‌کند و از اینروست که می‌تواند بکار گرفته شده یا در هر زمان اجرای برنامه آن را اصلاح کرد. کلاس حاوی توابع عضو **setCourseName**، **getCourseName** و **displayMessage** است. تابع عضو **setCourseName** نام دوره را در عضو داده **GradeBook** ذخیره می‌سازد. تابع عضو **getCourseName** نام دوره را از عضو داده بدست می‌آورد. تابع عضو **displayMessage** که هیچ پارامتری ندارد، پیغام خوش‌آمدگویی را که شامل نام دوره است، به نمایش در می‌آورد. با این همه، همانطوری که مشاهده می‌کنید، در حال حاضر تابع، نام دوره را با فراخوانی تابع دیگری در همان کلاس بدست می‌آورد، تابع **getCourseName**.

```
1  // Fig. 3.5: fig03_05.cpp
2  // Define class GradeBook that contains a courseName data member
3  // and member functions to set and get its value;
4  // Create and manipulate a GradeBook object.
5  #include <iostream>
6  using std::cout;
7  using std::cin;
8  using std::endl;
```

```
9
10 #include <string> // program uses C++ standard string class
11 using std::string;
12 using std::getline;
13
14 // GradeBook class definition
15 class GradeBook
16 {
17 public:
18    // function that sets the course name
19    void setCourseName( string name )
20    {
21       courseName = name; // store the course name in the object
22    } // end function setCourseName
23
24    // function that gets the course name
25    string getCourseName()
26    {
27       return courseName; // return the object's courseName
28    } // end function getCourseName
29
30    // function that displays a welcome message
31    void displayMessage()
32    {
33       // this statement calls getCourseName to get the
34       // name of the course this GradeBook represents
35      cout << "Welcome to the grade book for\n" << getCourseName() << "!"
36          << endl;
37    } // end function displayMessage
38 private:
39    string courseName; // course name for this GradeBook
40 }; // end class GradeBook
41
42 // function main begins program execution
43 int main()
44 {
45    string nameOfCourse; // string of characters to store the course name
46    GradeBook myGradeBook; // create a GradeBook object named myGradeBook
47
48    // display initial value of courseName
49    cout << "Initial course name is: " << myGradeBook.getCourseName()
50       << endl;
51
52    // prompt for, input and set course name
53    cout << "\nPlease enter the course name:" << endl;
54    getline( cin, nameOfCourse ); // read a course name with blanks
55    myGradeBook.setCourseName( nameOfCourse ); // set the course name
56
57    cout << endl; // outputs a blank line
58    myGradeBook.displayMessage(); // display message with new course name
59    return 0; // indicate successful termination
60 } // end main
```

```
Initial course name is:

Please enter the course name:
CS101 Introduction to C++ Programming

Welcome to the grade book for
CS101 Introduction to C++ Programming!
```

شکل ۵-۳ | تعریف و تست کلاس GradeBook با یک عضو داده و توابع set و get.

برنامه‌نویسی ایده‌ال

یک خط خالی مابین تعریف تابع عضو قرار دهید تا خوانایی برنامه افزایش پیدا کند.

معمولاً یک مدرس بیش از یک دوره را تدریس می‌کند که هر دوره دارای نام متعلق بخود است. در خط 39، متغیر **CourseName** از نوع رشته اعلان شده است. بدلیل اینکه متغیر اعلان شده در درون تعریف کلاس اعلان شده (خطوط 40-15) اما در خارج از بدنه تعاریف توابع عضو جای دارد (خطوط 22-19، 28-25 و 37-31)، خط 39 اعلانی برای یک عضو داده است. هر نمونه‌ای (شی) از کلاس **GradeBook** حاوی یک کپی از هر عضو داده کلاس خواهد بود. برای مثال، اگر دو شی **GradeBook** وجود داشته باشد، هر شی دارای یک کپی از **CourseName** متعلق بخود خواهد بود، همانطوری که در مثال برنامه شکل ۷-۳ شاهد آن خواهید بود. مزیت ایجاد **CourseName** بصورت یک عضو داده در این است که تمام توابع عضو یک کلاس (در این مورد، **GradBook**) می‌توانند در هر عضو داده موجود که در تعریف کلاس وجود دارد، دستکاری کنند (در این مورد، **CourseName**).

تصریح‌کننده دسترسی public و private

اکثر اعلان‌های اعضای داده پس از برچسب تصریح‌کننده دسترسی **private:** ظاهر می‌شوند (خط 38). همانند **public**، کلمه کلیدی **private** یک تصریح‌کننده است. متغیرها یا توابع اعلان شده پس از **private** (و قبل از تصریح‌کننده دسترسی بعدی) فقط در توابع عضو کلاسی که در آن اعلان شده‌اند، در دسترس خواهند بود. بنابر این، عضو داده **CourseName** فقط می‌تواند در توابع عضو **setCourseName**، **getCourseName** و **displayMessage** از کلاس **GradeBook** بکار گرفته شود. بدلیل اینکه عضو داده **CourseName** بصورت **private** اعلان شده است، نمی‌تواند توسط توابع خارج از کلاس (همانند **main**) یا توابع عضو کلاس‌های دیگر در برنامه بکار گرفته شود. مبادرت به دسترسی عضو داده **courseName** در یکی از نقاط این برنامه با عبارتی همانند **myGradeBook.courseName** خطای کامپایل تولید کرده و پیغامی مشابه

```
cannot access private member declared in class 'GradeBook'
```

به نمایش در می‌آید.

مهندسی نرم‌افزار

*بعنوان یک قانون، اعضای داده بایستی بصورت **private** و توابع عضو بصورت **public** اعلان شوند. (مشاهده خواهید کرد در صورتیکه توابع عضو خاصی که فقط از طریق توابع عضو دیگر موجود در کلاس در دسترس قرار می‌گیرند، اگر بصورت **private** اعلان شوند، مناسب خواهند بود.)*

خطای برنامه‌نویسی

*اقدام یک تابع، که عضوی از یک کلاس خاص نیست، به دسترسی به یک عضو **private** آن کلاس، خطای کامپایل بدنبال خواهد داشت.*

دسترسی پیش‌فرض برای اعضای کلاس حالت **private** است، از اینرو تمام اعضای قرار گرفته پس از سرآیند و قبل از اولین تصریح‌کننده دسترسی بصورت **private** خواهند بود. امکان دارد تصریح‌کننده‌های دسترسی **public** و **private** تکرار شوند، اما اینکار ضرورتی ندارد و می‌تواند سبب سردرگمی شود.

برنامه‌نویسی ایده‌ال

*علیرغم این واقعیت که تصریح‌کننده‌های دسترسی **public** و **private** می‌توانند تکرار شده و جابجا نوشته شوند، اما سعی کنید تمام لیست عضوهای **public** یک کلاس را در یک گروه و تمام عضوهای **private** را در گروه دیگری قرار دهید. در این حالت توجه سرویس‌گیرنده بر روی واسط **public** کلاس متمرکز می‌شود، بجای اینکه توجه آن به پیاده‌سازی کلاس معطوف شود.*

برنامه‌نویسی ایده‌ال

*اگر می‌خواهید لیست اعضای **private** را در ابتدای تعریف کلاس قرار دهید، بصورت صریح از کلمه **private** استفاده کنید علیرغم اینکه **private** انتخاب پیش‌فرض است. در اینحالت وضوح برنامه افزایش می‌یابد.*

اعلان اعضای داده با تصریح‌کننده دسترسی **private**، بعنوان پنهان‌سازی داده شناخته می‌شود. زمانیکه برنامه مبادرت به ایجاد (نمونه‌سازی) از یک شی کلاس **GradeBook** می‌کند، عضو داده **courseName** در شی کپسوله (پنهان) شده و فقط می‌تواند توسط توابع عضو آن کلاس در دسترس قرار گیرد. در کلاس **GradeBook**، توابع عضو **setCourseName** و **getCourseName** مبادرت به دستکاری مستقیم عضو داده **courseName** می‌کنند (اگر لازم باشد **diaplayMessage** هم می‌تواند این کار را انجام دهد).

مهندسی نرم‌افزار

*در فصل دهم خواهید آموخت که توابع و کلاس‌های اعلان شده توسط یک کلاس حالت دوست **(friend)** پیدا کرده و می‌توانند به اعضای **private** کلاس دسترسی پیدا کنند.*

اجتناب از خطا

*با ایجاد اعضای داده کلاس بصورت **private** و توابع عضو کلاس بصورت **public** کار خطایابی آسانتر می‌شود چرا که دستکاری‌کننده‌های داده محلی هستند.*

توابع عضو setCourseName و getCourseName

تابع عضو **setCourseName** تعریف شده در خطوط 22-19 به هنگام اتمام وظیفه خود هیچ مقداری برگشت نمی‌دهد و از اینرو نوع برگشتی آن **void** است. تابع عضو یک پارامتر دریافت می‌کند، **name**، که نشان‌دهنده نام دوره‌ای است که به آن بعنوان یک آرگومان ارسال خواهد شد (خط 55 از **main**).

خط 21 مبادرت به تخصیص **name** به عضو داده **courseName** می‌کند. در این مثال، **setCourseName** مبادرت به اعتبارسنجی نام دوره نمی‌کند (بررسی فرمت نام دوره با یک الگوی خاص). برای نمونه فرض کنید که دانشگاهی می‌تواند از اسامی دوره تا 25 کاراکتر یا کمتر چاپ بگیرد. در این مورد، مایل هستیم کلاس **GradeBook** ما را مطمئن سازد که عضو داده **courseName** هرگز بیش از 25 کاراکتر نداشته باشد. در بخش ۱۰-۳ با تکنیک‌های اولیه اعتبارسنجی آشنا خواهید شد.

تابع عضو **getCourseName** تعریف شده در خطوط 28-25 یک شی خاص از **GradeBook** بنام **courseName** را برگشت می‌دهد. تابع عضو دارای یک لیست پارامتری خالی است و از اینرو برای انجام وظیفه خود نیازی به اطلاعات اضافی ندارد. تابع مشخص می‌سازد که یک رشته برگشت خواهد داد. زمانیکه تابعی که نوع برگشتی آن بجز **void** است فراخوانی می‌شود و وظیفه خود را انجام می‌دهد، تابع نتیجه‌ای را به فراخوان خود برگشت می‌دهد. برای مثال، هنگامی که به سراغ یک ATM می‌روید و تقاضای نمایش میزان موجودی خود را می‌کنید، انتظار دارید ATM مقداری که نشاندهنده میزان موجودی است به شما برگشت دهد. به همین ترتیب، هنگامی که عبارتی تابع عضو **getCourseName** از یک شی **GradeBook** را فراخوانی می‌کند، عبارت انتظار دریافت نام دوره را دارد (در این مورد، یک رشته است که توسط نوع برگشتی تابع مشخص گردیده است). اگر تابعی بنام **square** داشته باشید که مربع آرگومان خود را برگشت می‌دهد، عبارت

```
int results = square(2);
```

مقدار **4** را از تابع **square** برگشت داده و متغیر **result** را با **4** مقداردهی اولیه می‌کند. اگر تابعی بنام **maximum** داشته باشید که بزرگترین عدد را از بین سه آرگومان خود برگشت می‌دهد، عبارت

```
int biggest = maximum(27,114,41);
```

عدد **114** را از تابع **maximum** برگشت داده و متغیر **biggest** با عدد **114** مقداردهی اولیه می‌شود.

خطای برنامه‌نویسی

فراموش کردن مقدار برگشتی از تابعی که مقدار برگشتی برای آن در نظر گرفته شده است، خطای کامپایل بدنبال خواهد داشت.

توجه کنید که عبارات قرار گرفته در خطوط 21 و 27 هر یک از متغیر **courseName** (خط 39) استفاده کرده‌اند، بدون اینکه این متغیر در توابع عضو اعلان شده باشد می‌توانیم از **courseName** در توابع عضو کلاس **GradeBook** استفاده کنیم چرا که **courseName** یک عضو داده از کلاس است. همچنین

توجه نمائید که ترتیب تعریف توابع عضو تعیین‌کننده ترتیب فراخوانی در زمان اجرا نیستند از اینرو تابع عضو **getCourseName** می‌تواند قبل از تابع عضو **setCourseName** تعریف شود.

تابع عضو displayMessage

تابع عضو **displayMessage** در خطوط 37-31 هیچ نوع داده‌ای را پس از کامل کردن وظیفه خود برگشت نمی‌دهد، از اینروست که نوع برگشتی آن **void** تعریف شده است. تابع، پارامتری دریافت نمی‌کند، بنابر این لیست پارامتری آن خالی است. خطوط 36-35 پیغام خوش‌آمدگویی را که حاوی مقدار عضو داده **courseName** است در خروجی چاپ می‌کنند. خط 35 مبادرت به فراخوانی تابع **getCourseName** برای بدست آوردن مقداری از **courseName** می‌کند. توجه نمائید که تابع عضو **displayMessage** هم قادر به دسترسی مستقیم به عضو داده **courseName** است. در مورد اینکه چرا از فراخوانی تابع عضو **getCourseName** برای بدست آوردن مقدار **courseName** استفاد کرده‌ایم، توضیح خواهیم داد.

تست کلاس GradeBook

تابع **main** (خطوط 60-43) یک شی از کلاس **GradeBook** ایجاد کرده و از توابع عضو آن استفاده می‌کند. در خط 46 یک شی **GradeBook** بنام **myGradeBook** ایجاد شده است. خطوط 50-49 نام دوره اولیه را با فراخوانی تابع عضو **getCourseName** به نمایش در می‌آورند. توجه کنید که اولین خط خروجی نشاندهنده نام دوره نمی‌باشد، چرا که عضو داده **courseName** در ابتدای کار تهی است. بطور پیش‌فرض، مقدار اولیه یک رشته، رشته تهی است، رشته‌ای که حاوی هیچ کاراکتری نمی‌باشد. زمانیکه یک رشته تهی به نمایش درآید، چیزی بر روی صفحه نمایش ظاهر نمیشود.

خط 53 از کاربر می‌خواهد که نام دوره را وارد سازد. متغیر محلی **nameOfCourse** اعلان شده در خط 45، با نام دوره وارد شده توسط کاربر مقداردهی می‌شود که از فراخوانی تابع **getline** بدست می‌آید (خط 54). خط 55 تابع عضو **setCourseName** را فراخوانی کرده و **nameOfCourse** را بعنوان آرگومان تابع بکار می‌گیرد. به هنگام فراخوانی تابع، مقدار آرگومان به پارامتر **name** (خط 19) از تابع عضو **setCourseName** کپی می‌گردد (خطوط 22-19). سپس مقدار پارامتر به عضو داده **courseName** تخصیص می‌یابد (خط 21). خط 57 یک خط خالی در خروجی قرار داده، سپس خط 58 تابع عضو **displayMessage** را برای نمایش پیغام خوش‌آمدگویی به همراه نام دوره فراخوانی می‌کند.

مهندسی نرم‌افزار با توابع set و get

اعضای داده **private** یک کلاس فقط قادر به دستکاری شدن از طریق توابع عضو آن کلاس هستند (و «دوستان» آن کلاس که در فصل دهم با آنها آشنا خواهید شد). بنابر این سرویس‌گیرنده یک شی،

(یعنی هر کلاس یا تابعی که مبادرت به فراخوانی توابع عضو از خارج شی می‌کند)، توابع عضو **public** کلاس را برای دریافت سرویس‌های کلاس برای شی‌های خاصی از کلاس فراخوانی می‌کند. به همین دلیل است که عبارات موجود در تابع **main** (برنامه شکل ۵-۳، خطوط 60-43) مبادرت به فراخوانی توابع عضو **setCourseName**، **getCourseName** و **displayMessage** موجود در شی **GradeBook** می‌کند. غالباً توابع عضو **public** کلاس‌ها به سرویس‌گیرنده‌ها اجازه تخصیص مقادیر به (set) یا دریافت مقادیر از (get) اعضای داده **private** را می‌دهند. نیازی نیست که اسامی این توابع عضو حتماً با set یا get شروع شوند، اما این روش نام‌گذاری مرسوم است. در این مثال، تابع عضوی که مبادرت به تنظیم (تخصیص مقادیر) عضو داده **courseName** می‌کند، **setCourseName** نام دارد و تابع عضوی که مقداری از عضو داده **courseName** بدست می‌آورد، تابع **getCourseName** نامیده می‌شود. توجه کنید که گاهاً توابع set، بنام *mutators* شناخته می‌شوند (چرا که مبادرت به تغییر یا اصلاح مقادیر می‌کنند)، و توابع get بنام *accessors* نامیده می‌شوند (چرا که به مقادیر دسترسی پیدا می‌کنند).

بخاطر دارید که اعلان اعضای داده با تصریح‌کننده دسترسی **private** موجب پنهان‌سازی داده می‌شود. تدارک دیدن توابع *set* و *get* به سرویس‌گیرنده‌های کلاس اجازه دسترسی به داده‌های پنهان شده را می‌دهند، اما بصورت غیرمستقیم. سرویس‌گیرنده از مبادرت خود به اصلاح یا بدست آوردن داده یک شی اطلاع دارد، اما از اینکه شی چگونه این عملیات را انجام می‌دهد، اطلاعی ندارد. در برخی از موارد امکان دارد کلاسی به یک روش مبادرت به عرضه داده در محیط داخلی نماید، در حالیکه همان داده را به روش مختلفی در اختیار سرویس‌گیرنده‌ها قرار می‌دهد. برای مثال، فرض کنید کلاس **Clock** زمانی از روز را بصورت یک عضو داده **time** و از نوع **int** و **private** عرضه می‌کند که تعداد ثانیه‌های از نیمه‌شب را ذخیره می‌سازد. با این همه، زمانیکه سرویس‌گیرنده‌ای تابع عضو **getTime** از شی **Clock** را فراخوانی می‌کند، زمان برحسب ساعت، دقیقه و ثانیه در یک رشته با فرمت "HH:MM:SS" به وی برگشت داده می‌شود. به همین ترتیب، فرض کنید کلاس **Clock** یک تابع set بنام **setTime** تدارک دیده که یک رشته پارامتری با فرمت "HH:MM:SS" دریافت می‌نماید. با استفاده از قابلیت‌های عرضه شده در فصل هیجدهم، تابع **setTime** قادر به تبدیل این رشته به تعداد ثانیه‌ها است، که تابع آن را در عضو داده **private** ذخیره سازد. همچنین تابع *set* می‌تواند به بررسی اعتبار مقدار دریافتی پرداخته و اعتبار آن را تایید یا لغو کند (مثلاً "12:30:43" معتبر بوده اما "42:85:70" معتبر نیست). توابع *set* و *get* به سرویس‌گیرنده امکان تعامل با یک شی را فراهم می‌آورند، اما داده **private** (خصوصی) شی همچنان بصورت کپسوله (پنهان) و ایمن در خودش نگهداری می‌شود.

همچنین توابع *set* و *get* یک کلاس می‌توانند توسط سایر توابع عضو موجود در درون کلاس به منظور دستکاری کردن داده **private** کلاس بکار گرفته شوند، اگر چه این توابع عضو می‌تواند بصورت مستقیم به داده **private** دسترسی پیدا کنند. در برنامه شکل ۵-۳، توابع عضو **getCourseName** و **setCourseName** از نوع توابع عضو **public** هستند، از اینرو در دسترس گیرنده‌های کلاس و همچنین خود کلاس قرار دارند. تابع عضو **displayMessage** تابع عضو **getCourseName** را برای بدست آوردن مقدار عضو داده **courseName** برای نمایش آن، فراخوانی می‌کند، با اینکه خود **displayMessage** می‌تواند بصورت مستقیم به **courseName** دسترسی پیدا کند. دسترسی به عضو داده از طریق تابع *get* آن، سبب ایجاد یک کلاس بهتر و پایدارتر می‌شود (کلاسی که نگهداری آن آسان بوده و کمتر از کار می‌افتد). اگر تصمیم به تغییر عضو داده **courseName** بگیریم، ساختار تعریف‌کننده **displayMessage** نیازمند اصلاح نخواهد بود، فقط بدنه توابع *get* و *set* که مستقیماً عضو داده را دستکاری می‌کنند نیاز به تغییر خواهند داشت. برای مثال، فرض کنید که می‌خواهیم نام دوره را در دو عضو داده عرضه کنیم، **courseName** (مثلاً "CS101") و **courseTitle** (مثلاً "Introduction to C++ Programming"). هنوز هم تابع **displayMessage** می‌تواند با یک فراخوانی تابع عضو **getCourseName** نام کامل دوره را بدست آورده و بعنوان بخشی از پیغام خوش‌آمدگویی به نمایش در آورد. در اینحالت، تابع **getCourseName** نیازمند ایجاد و برگشت دادن یک رشته **حاوی** **courseNumber** و بدنبال آن **courseTitle** است. در ادامه تابع عضو **displayMessage** عنوان کامل دوره "CS101 Introduction to C++ Programming" را به نمایش در خواهد آورد، چرا که از تغییرات صورت گرفته بر روی اعضای داده کلاس به دور مانده است. مزایای فراخوانی تابع *set* از یک تابع عضو دیگر کلاس به هنگام بحث اعتبارسنجی در بخش ۱۰-۳ بیشتر آشکار خواهد شد.

برنامه‌نویسی ایده‌ال

سعی کنید همیشه میزان تاثیرات در اعضای داده کلاس را با استفاده از توابع set و get در سطح محلی نگهداری کنید.

مهندسی نرم‌افزار

نوشتن برنامه‌هایی که درک و نگهداری آنها آسان باشد، مهم است. تغییرات همیشه رخ می‌دهند. بعنوان برنامه‌نویس باید انتظار داشته باشید که کدهای نوشته شده توسط شما زمانی به تغییر نیاز پیدا خواهند کرد.

مهندسی نرم‌افزار

*طراح کلاس نیازی به تدارک دیدن توابع set یا get برای هر ایتم داده **private** ندارد، این موارد فقط در صورت نیاز و شرایط مقتضی در نظر گرفته شوند. اگر سرویسی برای کد سرویس‌گیرنده مناسب باشد، باید آن سرویس در واسط **public** کلاس وارد گردد.*

دیاگرام UML کلاس GradeBook با یک داده عضو و توابع set و get

شکل ۶-۳ حاوی دیاگرام کلاس UML به روز شده برای نسخه‌ای از کلاس **GradeBook** بکار رفته در برنامه ۵-۳ است. این دیاگرام مبادرت به مدل‌سازی داده عضو **courseName** بعنوان یک صفت در بخش میانی کلاس کرده است. UML اعضای داده را در لیستی که در آن نام صفت، یک کولن و نوع صفت قرار گرفته عرضه می‌کند. نوع UML صفت **courseName**، نوع **String** است که متناظر با **string** در C++ می‌باشد. عضو داده **courseName** در C++ حالت **private** دارد و از اینرو در دیاگرام کلاس با یک علامت منفی (-) در مقابل نام صفت مشخص شده است. علامت منفی در UML معادل با تصریح‌کننده دسترسی **private** در C++ است. کلاس **GradeBook** حاوی سه تابع عضو **public** است، از اینرو در لیست دیاگرام کلاس این سه عملیات در بخش تحتانی یا سوم جای گرفته‌اند. بخاطر دارید که نماد جمع (+) قبل نام هر عملیات نشان می‌دهد که عملیات در C++ حالت **public** دارد. عملیات **setCourseName** دارای یک پارامتر **String** بنام **name** است. در UML نوع برگشتی از یک عملیات با قرار دادن یک کولن و نوع برگشتی پس از پرانتزهای نام عملیات مشخص می‌شود. تابع عضو **getCourseName** از کلاس **GradeBook** (شکل ۵-۳) دارای نوع **string** برگشتی در C++ است، بنابر این دیاگرام کلاس نوع برگشتی **String** را در UML به نمایش درآورده است. توجه نمائید که عملیات‌های **setCourseName** و **displayMessage** مقداری برگشت نمی‌دهند (نوع برگشتی آنها **void** است)، از اینرو در دیاگرام کلاس UML نوع برگشتی از پرانتزها برای آنها در نظر گرفته نشده است. UML از **void** همانند C++ در زمانیکه تابع مقداری برگشت نمی‌دهد، استفاده نمی‌کند.

شکل ٦-٣ | دیاگرام کلاس UML برای کلاس GradeBook با یک صفت private برای courseName و عملیات‌های public برای توابع setCourseName، getCourseName و displayMessage

۳-۷ مقداردهی اولیه شی‌ها با سازنده‌ها

همانطوری که از بخش ۶-۳ بخاطر دارید، زمانیکه یک شی از کلاس **GradeBook** ایجاد شد (شکل ۵-۳) عضو داده آن یعنی **courseName** بطور پیش‌فرض با یک رشته تهی مقداردهی اولیه شده بود. اگر بخواهیم به هنگام ایجاد یک شی از **GradeBook** نام دوره‌ای تدارک دیده شود، چه کاری باید انجام داد؟ هر کلاسی که اعلان می‌کنید می‌تواند یک سازنده (*constructor*) داشته باشد که می‌توان با استفاده از آن مبادرت به مقداردهی اولیه یک شی از کلاس به هنگام ایجاد شی کرد. سازنده یک تابع عضو ویژه است که بایستی همنام با نام کلاس تعریف شده باشد، از اینروست که کامپایلر می‌تواند آن را از دیگر توابع عضو کلاس تشخیص دهد. مهمترین تفاوت موجود مابین سازنده‌ها و توابع دیگر در این است که سازنده‌ها نمی‌تواند مقدار برگشت دهند، بنابر این نمی‌توانند نوع برگشتی داشته باشند (حتی **void**).

معمولاً، سازنده‌ها بصورت **public** اعلان می‌شود. غالباً کلمه "constructor" در برخی از نوشته‌ها بصورت خلاصه شده "ctor" بکار گرفته می‌شود، که استفاده از آن را ترجیح نداده‌ایم.

C++ نیازمند فراخوانی یک سازنده برای هر شی است که ایجاد می‌شود، در چنین حالتی مطمئن خواهیم بود که شی قبل از اینکه توسط برنامه بکار گرفته شود، بدرستی مقداردهی اولیه شده است. فراخوانی سازنده به هنگام ایجاد شی، بصورت غیرصریح یا ضمنی انجام می‌شود. در هر کلاسی که بصورت صریح سازنده‌ای را مشخص نکرده است، کامپایلر یک *سازنده پیش‌فرض* تدارک می‌بیند، این سازنده دارای پارامتر نمی‌باشد. برای مثال، زمانیکه خط 46 از برنامه شکل ۵-۳ یک شی **GradeBook** ایجاد می‌کند، سازنده پیش‌فرض فراخوانی می‌شود، چرا که در اعلان **myGradeBook** بطور صریح هیچ آرگومان سازنده‌ای مشخص نشده است. سازنده پیش‌فرض تدارک دیده شده از سوی کامپایلر یک شی **GradeBook** بدون هیچ گونه مقادیر اولیه برای اعضای داده شی ایجاد می‌کند. [نکته: برای اعضای داده که شی‌های از سایر کلاس‌ها هستند، سازنده پیش‌فرض بصورت ضمنی هر سازنده پیش‌فرض عضو داده را برای اطمینان از اینکه اعضاء داده به درستی مقداردهی اولیه شده‌اند، فراخوانی می‌کند. در واقع به این دلیل است که عضو داده **courseName** از نوع رشته (در برنامه شکل ۵-۳) با یک رشته تهی مقداردهی اولیه شده است. سازنده پیش‌فرض برای کلاس **string** مبادرت به تنظیم یک مقدار رشته‌ای با رشته تهی می‌کند، در بخش ۳-۱۰ مطالب بیشتری در ارتباط با مقداردهی اولیه اعضای داده که شی‌های از کلاس‌های دیگر هستند خواهید آموخت.]

در مثال برنامه شکل ۷-۳، نام یک دوره را به هنگام ایجاد یک شی از **GradeBook** مشخص کرده‌ایم (خط 49). در این مورد، آرگومان "CS101 Introduction to C++ Programming" به سازنده شی **GradeBook** ارسال می‌شود (خطوط 20-17) و مبادرت به مقداردهی اولیه **courseName** می‌نماید. برنامه شکل ۷-۳ یک کلاس **GradeBook** اصلاح شده تعریف کرده که حاوی یک سازنده با یک پارامتر رشته‌ای است که نام دورهٔ اولیه را دریافت می‌کند.

```
1  // Fig. 3.7: fig03_07.cpp
2  // Instantiating multiple objects of the GradeBook class and using
3  // the GradeBook constructor to specify the course name
4  // when each GradeBook object is created.
5  #include <iostream>
6  using std::cout;
7  using std::endl;
8
9  #include <string> // program uses C++ standard string class
10 using std::string;
11
12 // GradeBook class definition
```

```
13 class GradeBook
14 {
15 public:
16   // constructor initializes courseName with string supplied as argument
17    GradeBook( string name )
18    {
19       setCourseName( name );//call set function to initialize courseName
20    } // end GradeBook constructor
21
22    // function to set the course name
23    void setCourseName( string name )
24    {
25       courseName = name; // store the course name in the object
26    } // end function setCourseName
27
28    // function to get the course name
29    string getCourseName()
30    {
31       return courseName; // return object's courseName
32    } // end function getCourseName
33
34    // display a welcome message to the GradeBook user
35    void displayMessage()
36    {
37       // call getCourseName to get the courseName
38       cout << "Welcome to the grade book for\n" << getCourseName()
39          << "!" << endl;
40    } // end function displayMessage
41 private:
42    string courseName; // course name for this GradeBook
43 }; // end class GradeBook
44
45 // function main begins program execution
46 int main()
47 {
48    // create two GradeBook objects
49    GradeBook gradeBook1( "CS101 Introduction to C++ Programming" );
50    GradeBook gradeBook2( "CS102 Data Structures in C++" );
51
52    // display initial value of courseName for each GradeBook
53    cout <<"gradeBook1 created for course:"<< gradeBook1.getCourseName()
54       <<"\ngradeBook2 created for course:"<< gradeBook2.getCourseName()
55       << endl;
56    return 0; // indicate successful termination
57 } // end main
```

```
gradeBook1 created for course : CS101 Introduction to C++ Programming
gradeBook2 created for course : CS102 Data Structures in C++
```

شکل ۷-۳ | نمونه‌سازی شی‌های مضاعف از کلاس GradeBook و استفاده از سازنده GradeBook برای مشخص کردن نام دوره به هنگام ایجاد هر شی GradeBook.

تعریف سازنده

در خطوط 20-17 برنامه شکل ۷-۳ یک سازنده برای کلاس **GradeBook** تعریف شده است. توجه کنید که سازنده دارای نام مشابه همانند کلاس خود یعنی **GradeBook** است. یک سازنده توسط لیست پارامتری، داده مورد نیاز برای انجام وظیفه خود را مشخص می‌سازد. زمانیکه یک شی جدید ایجاد

می‌کنید، این داده را در درون پرانتزهای قرار گرفته پس از نام شی قرار می‌دهید (همانند خطوط 50-49). خط 17 بر این نکته دلالت دارد که سازنده کلاس **GradeBook** دارای یک پارامتر رشته‌ای بنام **name** است. دقت کنید که در خط 17 نوع برگشتی مشخص نشده است، چرا که سازنده‌ها نمی‌توانند مقدار برگشت دهند (حتی **void**).

خط 19 در بدنه سازنده مبادرت به ارسال پارامتر **name** سازنده به تابع عضو **setCourseName** می‌کند که مقداری به عضو داده **courseName** تخصیص می‌دهد. تابع عضو **setCourseName** (خطوط 26-23) فقط مبادرت به تخصیص پارامتر **name** خود به عضو داده **courseName** می‌کند، بنابر این ممکن است تعجب کنید که چرا زحمت فراخوانی **setCourseName** را در خط 19 به خود داده‌ایم، در حالیکه سازنده بطور مشخص عملیات تخصیص **courseName=name** را انجام می‌دهد. در بخش ۱۰-۳، مبادرت به اصلاح **setCourseName** برای انجام عملیات اعتبارسنجی خواهیم کرد. در این بخش است که مزیت فراخوانی **setCourseName** از سازنده آشکار خواهد شد. توجه نمائید که در سازنده (خط 17) و هم تابع **setCourseName** (خط 23) از پارامتری بنام **name** استفاده شده است. می‌توانید از اسامی پارامتری یکسان در توابع مختلف استفاده کنید چرا که پارامترها حالت محلی برای هر تابع دارند و سبب تداخل با دیگری نمی‌شوند.

تست کلاس GradeBook

خطوط 57-46 از برنامه شکل ۷-۳ حاوی تعریف تابع **main** است که مبادرت به تست کلاس **GradeBook** و اثبات مقداردهی اولیه شی **GradeBook** با استفاده از یک سازنده می‌کند. خط 49 در تابع **main** اقدام به ایجاد و مقداردهی اولیه یک شی **GradeBook** بنام **gradeBook1** می‌کند. زمانی که این خط از کد اجرا شود، سازنده **GradeBook** در خطوط 20-17 همراه با آرگومان "CS101 Introduction to C++ Programming" برای مقداردهی اولیه نام دوره **gradeBook1** فراخوانی می‌شود، البته این فراخوانی بصورت ضمنی و توسط C++ صورت می‌گیرد. خط 50 تکرار این فرآیند برای یک شی **GradeBook** بنام **gradeBook2** است، این بار، آرگومان "CS102 Data Structures in C++" برای مقداردهی اولیه نام دوره **gradeBook2** ارسال می‌شود. خطوط 54-53 از تابع عضو هر شی **getCourseName** برای بدست آوردن اسامی دوره و نمایش اینکه آنها در زمان ایجاد مقداردهی اولیه شده‌اند، استفاده کرده‌اند. خروجی برنامه تایید می‌کند که هر شی **GradeBook** از کپی عضو داده **courseName** متعلق بخود نگهداری کرده است.

دو روش برای تدارک دیدن یک سازنده پیش‌فرض برای یک کلاس

به هر سازنده‌ای که هیچ آرگومانی دریافت نکند، سازنده پیش‌فرض می‌گویند. یک کلاس به یکی از دو روش زیر سازندهٔ پیش‌فرض بدست می‌آورد:

۱- کامپایلر بطور ضمنی یک سازنده پیش‌فرض برای کلاسی که سازنده‌ای برای آن تعریف نشده است، ایجاد می‌کند. چنین سازنده‌ای مبادرت به مقداردهی اولیه اعضای داده کلاس نمی‌کند، اما برای هر عضو داده که شی از کلاس دیگری هستند، سازنده پیش‌فرض را فراخوانی می‌کند [نکته: معمولاً یک متغیر مقداردهی نشده حاوی یک مقدار «*اشغال*» است (مثلاً یک متغیر **int** مقداردهی نشده می‌تواند حاوی 858993460- باشد که این مقدار در بسیاری از برنامه‌ها برای این متغیر غلط می‌باشد).]

۲- برنامه‌نویس بصورت صریح مبادرت به تعریف سازنده‌ای نماید که آرگومانی دریافت نمی‌کند. چنین سازنده‌ای شروع به مقداردهی اولیه مطابق نظر برنامه‌نویس کرده و سازنده پیش‌فرض را برای هر داده عضو که شی از یک کلاس دیگر است فراخوانی می‌کند.

اگر برنامه‌نویس مبادرت به تعریف یک سازنده با آرگومان نماید، دیگر C++ بصورت ضمنی یک سازنده پیش‌فرض برای آن کلاس ایجاد نخواهد کرد. توجه نمائید که برای هر نسخه از کلاس **GradeBook** در برنامه‌های ۱-۳، ۳-۳ و ۵-۳ کامپایلر بصورت ضمنی یک سازنده پیش‌فرض تعریف کرده است.

اجتناب از خطا

بجز در مواردی که نیازی به مقداردهی اولیه اعضای داده کلاس ضروری نیست (تقریباً هیچ وقت)، یک سازنده پیش‌فرض در نظر بگیرید که حتماً اعضای داده کلاس را با مقادیر مناسب به هنگام ایجاد هر شی جدید مقداردهی اولیه نماید.

مهندسی نرم‌افزار

اعضای داده می‌توانند توسط سازنده یک کلاس مقداردهی اولیه شوند یا پس از ایجاد شی تنظیم گردند. با این وجود، از منظر مهندسی نرم‌افزار بهتر خواهد بود تا یک شی بطور کامل و قبل از اینکه سرویس‌گیرنده‌ای مبادرت به فراخوانی توابع عضو آن شی نماید، مقداردهی اولیه شده باشد. بطور کلی، نبایستی فقط متکی به کد سرویس‌گیرنده باشید که بدقت و بدرستی یک شی را مقداردهی نماید.

افزودن سازنده به دیاگرام کلاس UML کلاس GradeBook

دیاگرام کلاس UML در شکل ۸-۳ مبادرت به مدل کردن کلاس **GradeBook** برنامه ۷-۳ کرده است، که دارای یک سازنده با پارامتر **name** از نوع رشته است (نوع **String** در UML). همانند عملیات‌ها، UML مبادرت به مدل‌سازی سازنده‌ها در بخش سوم کلاس در دیاگرام کلاس می‌کند. برای

تمایز قائل شدن مابین یک سازنده از یک عملیات کلاس، UML مبادرت به قرار دادن کلمه "constructor" مابین گیومه (« و ») قبل از نام سازنده می‌کند. قرار دادن لیست سازنده کلاس قبل از دیگر عملیات‌ها در بخش سوم امری رایج است.

شکل ۸-۳ | دیاگرام کلاس UML نشان می‌دهد که کلاس GradeBook دارای یک سازنده با پارامتر name از نوع String است.

۸-۳ قرار دادن کلاس در یک فایل مجزا برای استفاده مجدد

تا آنجا که از منظر برنامه‌نویسی نیاز داشته باشیم به توسعه کلاس **GradeBook** ادامه خواهیم داد، از اینرو اجازه دهید تا وارد برخی از مباحث مهندسی نرم‌افزار شویم. یکی از مزایای تعریف دقیق یک کلاس در این است که به هنگام بسته (*package*) کردن صحیح، کلاس‌های ما می‌توانند توسط سایر برنامه‌نویسان در سرتاسر جهان بکار گرفته شوند (استفاده مجدد). برای مثال، کتابخانه استاندارد C++ دارای نوع **string** است که می‌توانیم از آن در هر برنامه C++ استفاده کنیم (استفاده مجدد). البته این کار را با وارد کردن فایل سرآیند <**string**> در برنامه انجام می‌دهیم.

متاسفانه، برنامه‌نویسانی که مایل به استفاده از کلاس **GradeBook** ما هستند، نمی‌تواند بسادگی و فقط با وارد کردن فایل از برنامه ۷-۳ به یک برنامه دیگر از آن استفاده کنند. همانطوری که در فصل دوم آموختید، تابع **main** اجرای هر برنامه‌ای را آغاز می‌کند و هر برنامه باید دارای یک تابع **main** باشد. اگر برنامه‌نویسان دیگر مبادرت به قرار دادن کد برنامه ۷-۳ نمایند، چمدان بزرگی بدست خواهند گرفت، تابع **main** ما، و برنامه آنها حاوی دو تابع **main** خواهد بود. زمانیکه مبادرت به کامپایل برنامه کنند، کامپایلر متوجه خطا خواهد شد، چرا که هر برنامه‌ای فقط می‌تواند یک تابع main داشته باشد. برای مثال، اگر مبادرت به کامپایل برنامه‌ای با دو تابع **main** در برنامه Mircrosoft Visual C++ .NET کنید، خطای زیر تولید می‌شود:

```
error C2084: function 'int main(void)' already has a body
```

هنگامی که کامپایلر سعی در کامپایل دومین تابع **main** می‌کند با خطا مواجه می‌شود. به همین ترتیب کامپایلر GNU C++ خطای زیر را تولید می‌کند:

```
redefinition of 'int main()'
```

این خطاها بر این نکته دلالت دارند که برنامه در حال حاضر دارای یک تابع **main** است، بنابر این، قرار دادن **main** در همان فایل با تعریف کلاس از اینکه بتوان از کلاس در سایر برنامه‌ها استفاده مجدد

کرد، ممانعت بعمل می‌آورد. در این بخش، به توضیح نحوه ایجاد کلاس **GradeBook** با هدف استفاده مجدد خواهیم پرداخت. روشی که در آن کلاس را در یک فایل مجزا از تابع **main** قرار می‌دهیم.

فایل‌های سرآیند

هر کدام یک از مثال‌های مطرح شده تا بدین مرحله متشکل از یک فایل **cpp.** بودند که بعنوان فایل کد-منبع نیز شناخته می‌شوند. این مثال‌ها حاوی تعریف کلاس **GradeBook** و یک تابع **main** بودند. به هنگام ایجاد یک برنامه شی‌گرای C++، تعریف کد منبع با قابلیت استفاده مجدد (همانند یک کلاس) در یک فایل که دارای پسوند فایل **h.** (فایل سرآیند) است، امری رایج می‌باشد. در برنامه‌ها از رهنمودهای پیش‌پردازنده **#include** برای وارد کردن فایل‌های سرآیند و برخوردار شدن از مزیت کامپونت‌های نرم‌افزاری با قابلیت استفاده مجدد کمک گرفته می‌شود، همانند نوع **string** تدارک دیده شده در کتابخانه استاندارد C++ و نوع‌های تعریف شده توسط کاربر مانند کلاس **GradeBook**.

در مثال بعدی، مبادرت به متمایز کردن کد برنامه از ۳-۷ با دو فایل GradeBook.h برنامه شکل ۳-۹ و fig03_10.cpp برنامه شکل ۳-۱۰ می‌کنیم. همانطوری که در فایل سرآیند شکل ۳-۹ مشاهده می‌کنید، این فایل فقط حاوی تعریف کلاس **GradeBook** (خطوط 41-11) و خطوط 8-3 است که به کلاس **GradeBook** اجازه استفاده از **endl**، **cout** و نوع **string** را می‌دهند. تابع **main** که از کلاس **GradeBook** استفاده می‌کند در فایل کد منبع fig03_10.cpp شکل ۳-۱۰ تعریف شده است، در خطوط 21-10. برای کمک به شما برای مهیا شدن برای کار با برنامه‌های بزرگ که در این کتاب و بازار کار با آنها مواجه خواهید شد، غالباً از یک فایل کد منبع متمایز یا جداگانه حاوی تابع **main** برای تست کلاس‌های خود استفاده می‌کنیم (به این برنامه، برنامه راه‌انداز یا درایور می‌گویند). بزودی با نحوه استفاده یک فایل کد منبع با **main** از تعریف کلاس موجود در یک فایل سرآیند در ایجاد شی‌های از یک کلاس آشنا خواهید شد.

```
1  // Fig. 3.9: GradeBook.h
2  // GradeBook class definition in a separate file from main.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <string> // class GradeBook uses C++ standard string class
8  using std::string;
9
10 // GradeBook class definition
11 class GradeBook
12 {
13 public:
14   // constructor initializes courseName with string supplied as argument
15    GradeBook( string name )
16    {
```

```
17       setCourseName( name );// call set function to initialize courseName
18    } // end GradeBook constructor
19
20    // function to set the course name
21    void setCourseName( string name )
22    {
23       courseName = name; // store the course name in the object
24    } // end function setCourseName
25
26    // function to get the course name
27    string getCourseName()
28    {
29       return courseName; // return object's courseName
30    } // end function getCourseName
31
32    // display a welcome message to the GradeBook user
33    void displayMessage()
34    {
35       // call getCourseName to get the courseName
36       cout << "Welcome to the grade book for\n" << getCourseName()
37          << "!" << endl;
38    } // end function displayMessage
39 private:
40    string courseName; // course name for this GradeBook
41 }; // end class GradeBook
```

شکل ۹-۳ | تعریف کلاس GradeBook.

وارد کردن فایل سرآیند که حاوی یک کلاس تعریف شده از سوی کاربر است

یک فایل سرآیند همانند GradeBook.h (برنامه شکل ۹-۳) نمی‌تواند برای شروع اجرای برنامه بکار گرفته شود، چرا که حاوی تابع **main** نمی‌باشد. اگر سعی در کامپایل و لینک خود GradeBook.h به منظور ایجاد یک برنامه اجرایی کنید، Microsoft Visual C++ .NET پیغام خطای لینکر را تولید خواهد کرد:

```
error LNK2019:unresolved external symbol _main referenced in
function _mainCRTStrartup
```

اگر در حال استفاده از GNU C++ بر روی لینوکس باشید، پیغام خطای لینکر بصورت زیر خواهد بود:

```
undefined reference to 'main'
```

این خطا بر این نکته دلالت دارد که لینکر قادر به یافتن تابع **main** برنامه نشده است. برای تست کلاس **GradeBook** تعریف شده در شکل ۹-۳ بایستی یک فایل کد منبع جداگانه حاوی یک تابع **main** (همانند شکل ۱۰-۳) بنویسید که مبادرت به نمونه‌سازی و استفاده از شی‌های کلاس کند.

از بخش ۴-۳ بخاطر دارید که کامپایلر از مفهوم نوع‌های بنیادین همانند **int** مطلع است، اما با **GradeBook** آشنا نیست چرا که این نوع یک نوع تعریف شده از سوی کاربر (برنامه‌نویس) است. در

واقع، کامپایلر حتی از کلاس‌های کتابخانه استاندارد C++ اطلاعی ندارد. برای کمک به کامپایلر برای اینکه متوجه نحوه استفاده از یک کلاس شود، بایستی بصورت صریح تعریف کلاس را در اختیار آن قرار دهیم. به همین دلیل است که برای مثال، به هنگام استفاده از نوع **string**، باید برنامه شامل فایل سرآیند <**string**> باشد. با انجام اینکار، کامپایلر قادر به تعیین میزان حافظه‌ای خواهد بود که باید برای هر شی از کلاس رزرو نماید و فراخوانی صحیح توابع عضو کلاس را تضمین می‌کند.

```
1  // Fig. 3.10: fig03_10.cpp
2  // Including class GradeBook from file GradeBook.h for use in main.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "GradeBook.h" // include definition of class GradeBook
8
9  // function main begins program execution
10 int main()
11 {
12    // create two GradeBook objects
13    GradeBook gradeBook1( "CS101 Introduction to C++ Programming" );
14    GradeBook gradeBook2( "CS102 Data Structures in C++" );
15
16    // display initial value of courseName for each GradeBook
17    cout <<"gradeBook1 created for course:"<< gradeBook1.getCourseName()
18       <<"\ngradeBook2 created for course:"<< gradeBook2.getCourseName()
19       << endl;
20    return 0; // indicate successful termination
21 } // end main
```

```
gradeBook1 created for course : CS101 Introduction to C++ Programming
gradeBook2 created for course : CS102 Data Structures in C++
```

شکل ۱۰-۳ | وارد کردن کلاس GradeBook از فایل GradeBook.h برای استفاده در main.

برای ایجاد شی‌های **gradeBook1** و **gradeBook2** در خطوط 14-13 برنامه ۱۰-۳، باید کامپایلر از سایز یک شی **GradeBook** مطلع باشد. در حالیکه شی‌ها بصورت مفهومی حاوی اعضای داده و توابع عضو هستند، شی‌های C++ فقط حاوی داده می‌باشند. کامپایلر فقط یک کپی از توابع عضو کلاس ایجاد کرده و آن کپی را در میان سایر شی‌های کلاس به اشتراک می‌گذارد. البته هر شی نیازمند کپی متعلق بخود از اعضای داده کلاس است، چرا که محتویات آنها می‌توانند با سایر شی‌ها بسیار تفاوت داشته باشند (همانند دو شی مختلف **BankAccount** (حساب بانکی) که دو عضو داده متفاوت **balance** یعنی موجودی دارند). با این وجود، کد تابع عضو تغییر پذیر نیست، از اینرو می‌تواند در میان تمام شی‌های کلاس به اشتراک گذاشته شود. بنابر این، سایز یک شی وابسته به میزان فضای حافظه مورد نیاز برای ذخیره‌سازی عضوهای داده کلاس است. با وارد کردن GradeBook.h در خط 7، به کامپایلر امکان دسترسی به اطلاعات مورد نیاز (شکل ۹-۳، خط 40) برای تعیین سایز یک شی **GradeBook** و تعیین

اینکه آیا شی‌های از آن کلاس بدرستی بکار گرفته شده‌اند یا خیر، داده می‌شود (در خطوط 14-13 و 17-18 شکل ۱۰-۳).

خط 7 به پیش‌پردازنده C++ دستور جایگزین رهنمود با یک کپی از محتویات GradeBook.h (تعریف کلاس **GradeBook**) قبل از کامپایل برنامه را صادر می‌کند. زمانیکه فایل کد منبع fig03_10.cpp کامپایل می‌شود، حاوی تعریف کلاس **GradeBook** بوده (بدلیل **#include**)، و کامپایلر قادر به تعیین نحوه ایجاد شی‌های **GradeBook** و مشاهده فراخوانی صحیح توابع عضو خواهد بود. اکنون که تعریف کلاس در یک فایل سرآیند (بدون تابع **main**) قرار دارد، می‌توانیم سرآیند را در هر برنامه‌ای که نیاز به استفاده مجدد از کلاس **GradeBook** دارد وارد کنیم.

نحوه یافتن فایل‌های سرآیند

توجه کنید که نام فایل سرآیند GradeBook.h در خط 7 شکل ۱۰-۳ در میان جفت گوتیشن (" ") بجای < > محدود شده است. معمولاً فایلهای کد منبع برنامه و فایل‌های سرآیند تعریف شده از سوی کاربر در همان شاخه قرار داده می‌شوند. زمانیکه پیش‌پردازنده با یک نام فایل سرآیند در میان گوتیشن‌ها مواجه می‌شود (مثل **"GradeBook.h"**)، مبادرت به مکان‌یابی فایل سرآیند در همان شاخه می‌کند که فایل حاوی **#include** در آن قرار دارد. اگر پیش‌پردازنده موفق به یافتن فایل سرآیند در آن شاخه نشود، شروع به جستجوی آن فایل در همان موقعیت(ها) همانند فایل‌های سرآیند کتابخانه استاندارد C++ می‌کند. زمانیکه پیش‌پردازنده با نام فایل سرآیند در میان < > مواجه شود (مثل **<iostream>**) فرض می‌کند که سرآیند بخشی از کتابخانه استاندارد C++ است و به شاخه‌ای که برنامه در آن قرار دارد نگاه نمی‌کند.

اجتناب از خطا

*برای اطمینان از اینکه پیش‌پردازنده قادر به مکان‌یابی صحیح فایل‌های سرآیند خواهد بود، بایستی رهنمود پیش‌پردازنده **#include** مبادرت به قرار دادن اسامی فایل‌های سرآیند تعریف شده از سوی کاربر در میان گوتیشن‌ها کند (همانند **"GradeBook.h"**) و اسامی فایل‌های سرآیند کتابخانه استاندارد **C++** را در میان < > قرار دهد (همانند **<iostream>**).*

مبحث مهندسی نرم‌افزار

اکنون که کلاس **GradeBook** در یک فایل سرآیند تعریف شده است، کلاس از قابلیت استفاده مجدد برخوردار شده است. متاسفانه، قرار دادن تعریف یک کلاس در یک فایل سرآیند همانند شکل ۹-۳ هنوز هم کل ساختار پیاده‌سازی کلاس را در دید سرویس‌گیرندگان کلاس قرار می‌دهد. GradeBook.h یک فایل متنی ساده است که هر کسی می‌تواند آن را باز کرده و بخواند. اصول مهندسی نرم‌افزار بر این نکته تاکید دارد که به هنگام استفاده از یک شی از کلاسی، کد سرویس‌گیرنده فقط نیاز به

فراخوانی توابع عضو مورد نیاز، اطلاع داشتن از تعداد آرگومان‌ها در هر تابع عضو و نوع برگشتی هر تابع عضو دارد. نیازی نیست که کد سرویس‌گیرنده از نحوه پیاده‌سازی توابع مطلع باشد.

اگر کد سرویس‌گیرنده از نحوه پیاده‌سازی یک کلاس مطلع باشد، امکان دارد برنامه‌نویس کد سرویس‌گیرنده براساس جزئیات ساختار پیاده‌سازی کلاس مبادرت به برنامه‌نویسی کد سرویس‌گیرنده کند. ایده‌آل نخواهد بود، اگر در پیاده‌سازی تغییری رخ دهد، سرویس‌گیرنده کلاس مجبور به تغییر در کد خود نباشد. با پنهان‌سازی جزئیات پیاده‌سازی کلاس، کار تغییر در ساختار کلاس آسانتر شده و احتمال تغییر در کد سرویس‌گیرنده‌های کلاس به حداقل می‌رسد. در بخش ۹-۳ شما را با نحوه تفکیک کلاس GradeBook به دو فایل آشنا خواهیم کرد که با اینکار

۱- کلاس قابلیت استفاده مجدد پیدا می‌کند

۲- سرویس گیرنده‌های کلاس از توابع عضو تدارک دیده شده توسط کلاس، نحوه فراخوانی و نوع برگشتی از آنها مطلع می‌شوند

۳- سرویس گیرنده‌ها از نحوه پیاده‌سازی توابع عضو کلاس مطلع نخواهند بود.

۹-۳ جداسازی واسط از پیاده‌سازی

در بخش قبلی، با نحوه افزودن قابلیت استفاده مجدد از نرم‌افزار توسط جداسازی تعریف از کد سرویس‌گیرنده که از کلاس استفاده می‌کند، آشنا شدید. اکنون بحث دیگری مطرح می‌کنیم که یکی از اصول کاربردی در مهندسی نرم‌افزار است، بحث *جداسازی واسط از پیاده‌سازی*.

واسط یک کلاس

واسط‌ها تعریف‌کننده روش‌های استاندارد در برقراری تعامل چیزهای همانند مردم و سیستم‌ها با یکدیگر هستند. برای‌مثال، کنترل‌های(دکمه‌های) رادیو نقش واسط مابین کاربران رادیو و کامپونت‌های داخلی آن بازی می‌کنند. کنترل‌ها به کاربران امکان انجام کارهای مشخصی را می‌دهند، کارهای همانند تغییر ایستگاه، تنظیم صدا و انتخاب ایستگاه‌های FM و AM. امکان دارد رادیوهای مختلف این عملیات‌ها را به روش‌های متفاوتی انجام دهند، برخی از دکمه‌های فشاری، برخی از صفات شاخص‌دار و برخی از دستورات صوتی پشتیبانی می‌کنند. واسط، تصریح‌کننده نوع عملیاتی است که رادیو اجازه انجام آن را به کاربر می‌دهد، اما نشان‌دهنده نحوه پیاده‌سازی عملیات در داخل رادیو نمی‌باشد.

به همین ترتیب، واسط یک کلاس توصیف‌کننده سرویس‌های (خدماتی) است که کلاس در اختیار سرویس‌گیرندگان خود قرار می‌دهد و نحوه تقاضای این سرویس‌ها را مشخص می‌سازد، اما چیزی از نحوه انجام کار به سرویس‌ها ارائه نمی‌کند. واسط یک کلاس متشکل از توابع عضو **public** که بعنوان سرویس‌های سراسری یا **public** کلاس هم شناخته می‌شوند، است. برای مثال، واسط کلاس

GradeBook (شکل ۹-۳) حاوی یک سازنده و توابع عضو **setCourseName**، **getCourseName** و **displayMessage** است. سرویس‌گیرنده **GradeBook** (مثل **main** در شکل ۱۰-۳) از این توابع برای تقاضای سرویسی از کلاس استفاده می‌کند. همانطوری که بزودی مشاهده خواهید کرد، می‌توانید واسط یک کلاس را با نوشتن تعریف کلاسی که فقط لیستی از اسامی توابع عضو، نوع برگشتی و نوع پارامترها دارد، مشخص کنید.

تفکیک واسط از پیاده‌سازی

در مثال‌های قبلی، هر تعریف کلاس حاوی تعاریف کاملی از توابع عضو **public** کلاس و اعلان‌های از اعضای داده **private** آن بود. با این وجود، از لحاظ مهندسی نرم‌افزار بهتر خواهد بود تا توابع عضو در خارج از تعریف کلاس، تعریف گردند، بنابر این جزئیات پیاده‌سازی آنها از دید کد سرویس‌گیرنده‌ها پنهان خواهند ماند. با اینکار مطمئن خواهید شد که برنامه‌نویسان نخواهند توانست براساس جزئیات پیاده‌سازی کلاس شما، مبادرت به نوشتن کد سرویس‌گیرنده کنند. اگر چنین کاری کنند، در صورتی که ساختار پیاده‌سازی کلاس را تغییر دهید، به احتمال زیاد کد آنها با شکست مواجه خواهد شد.

برنامه موجود در شکل‌های ۱۱-۳ الی ۱۳-۳ مبادرت به جداسازی واسط کلاس **GradeBook** از بخش پیاده‌سازی آن با تقسیم تعریف کلاس شکل ۹-۳ به دو فایل، فایل سرآیند GradeBook.h (شکل ۱۱-۳) که در آن کلاس **GradeBook** تعریف شده و فایل کد منبع GradeBook.cpp (شکل ۱۲-۳) که توابع عضو **GradeBook** در آن تعریف شده‌اند، می‌کند. بطور قراردادی تعاریف توابع عضو در یک فایل کد منبع همنام با نام فایل سرآیند کلاس جای داده می‌شوند، بجز اینکه پسوند فایل **cpp.** است. فایل کد منبع fig03_13.cpp (شکل ۱۳-۳) تعریف‌کننده تابع **main** است (کد سرویس‌گیرنده). کد و خروجی شکل ۱۳-۳ یکسان با شکل ۱۰-۳ است. شکل ۱۴-۳ نشاندهنده نحوه کامپایل این فایل برنامه از منظر برنامه‌نویس کلاس **GradeBook** و برنامه‌نویس کد سرویس‌گیرنده است، در ارتباط با این تصویر توضیحاتی ارائه خواهیم داد.

GradeBook.h: تعریف واسط کلاس با نمونه اولیه تابع

فایل سرآیند GradeBook.h (شکل ۱۱-۳) حاوی نسخه دیگری از تعریف کلاس **GradeBook** است (خطوط 17-9). این نسخه شبیه به نسخه موجود در شکل ۹-۳ است، اما تعاریف تابع در شکل ۹-۳ با *نمونه اولیه تابع* (*function prototype*) جایگزین شده‌اند (خطوط 15-12)، که توصیف کننده واسط **public** کلاس است بدون اینکه پیاده‌سازی تابع عضو کلاس را آشکار کرده باشد. نمونه اولیه تابع، اعلانی از تابع است که به کامپایلر نام تابع، نوع برگشتی آن و نوع پارمترهای آن را بیان می‌کند. دقت کنید که فایل سرآیند هنوز هم مشخص‌کننده عضو داده **private** کلاس است (خط 17). مجدداً بایستی

کامپایلر از اعضای داده مطلع باشد تا بتواند میزان حافظه رزرو شده برای هر شی از کلاس را تعیین کند. با وارد کردن فایل سرآیند GradeBook.h در کد سرویس‌گیرنده (خط 8 از شکل ۱۳-۳) این اطلاعات در اختیار کامپایلر قرار می‌گیرد.

نمونه اولیه تابع در خط 12 از شکل ۱۱-۳ بر این نکته دلالت دارد که سازنده نیازمند یک پارامتر رشته‌ای است. بخاطر دارید که سازنده‌ها دارای نوع برگشتی نیستند، از اینرو نوع برگشتی در نمونه اولیه تابع قرار داده نشده است. نمونه اولیه تابع عضو **setCourseName** در خط 13 نشان می‌دهد که **setCourseName** نیازمند یک پارامتر رشته‌ای بوده و مقداری برگشت نمی‌دهد (نوع برگشتی آن **void** است). نمونه اولیه تابع عضو **getCourseName** نشان می‌دهد که تابع نیازمند پارامتر نبوده و رشته برگشت می‌دهد (خط 14).

```
1  .// Fig. 3.11: GradeBook.h
2  .// GradeBook class definition. This file presents GradeBook's public
3  .// interface without revealing the implementations of GradeBook's member
4  .// functions, which are defined in GradeBook.cpp.
5  .#include <string> // class GradeBook uses C++ standard string class
6  .using std::string;
7  .
8  .// GradeBook class definition
9  .class GradeBook
10 .{
11 .public:
12 .   GradeBook( string ); // constructor that initializes courseName
13 .   void setCourseName( string ); // function that sets the course name
14 .   string getCourseName(); // function that gets the course name
15 .   void displayMessage(); // function that displays a welcome message
16 .private:
17 .   string courseName; // course name for this GradeBook
18 .}; // end class GradeBook
```

شکل ۱۱-۳ | تعریف کلاس GradeBook حاوی نمونه اولیه تابع که مشخص‌کننده واسط کلاس است.

در پایان، نمونه اولیه تابع عضو displayMessage قرار دارد که مشخص می‌کند این تابع نیازمند پارامتر نبوده و مقداری هم برگشت نخواهد داد (خط 15). این نمونه‌های اولیه تابع همانند سرآیندهای تابع متناظر در شکل ۹-۳ هستند، بجز اینکه اسامی پارامتر (که در نمونه‌های اولیه اختیاری هستند) در نظر گرفته نشده‌اند و اینکه نمونه اولیه هر تابع باید با یک سیمکولن خاتمه پذیرد.

خطای برنامه‌نویسی

فراموش کردن سیمکولن در انتهای نمونه اولیه یک تابع، خطای نحوی است.

برنامه‌نویسی ایده‌ال

اگر چه اسامی پارامتر در نمونه اولیه تابع اختیاری است (توسط کامپایلر نادیده گرفته می‌شوند) اما برخی از برنامه‌نویسان با هدف مستندسازی از این اسامی استفاده می‌کنند.

اجتناب از خطا

اسامی پارامترها در نمونه اولیه تابع (که توسط کامپایلر نادیده گرفته می‌شوند) می‌توانند در صورت استفاده اشتباه یا تداخل اسامی، مشکل‌ساز شوند. به همین دلیل برخی از برنامه‌نویسان ابتدا یک کپی از اولین خط تعریف تابع مربوطه تهیه و نمونه اولیه تابع را ایجاد (در صورتیکه منبع تابع در اختیار باشد)، سپس مبادرت به الحاق یک سیمکولن به انتهای هر نمونه اولیه می‌کنند.

GradeBook.cpp: تعریف توابع عضو در یک فایل کد منبع جداگانه

فایل کد منبع GradeBook.cpp شکل ۱۲-۳ تعریف‌کننده توابع عضو کلاس **GradeBook** است که در خطوط 15-12 از شکل ۱۱-۳ اعلان شده‌اند. تعریف تابع عضو در خطوط 34-11 قرار دارد و تقریباً با تعاریف موجود در خطوط 38-15 شکل ۹-۳ یکسان هستند.

توجه کنید که نام هر تابع عضو در سرآیندهای تابع (خطوط 11، 17، 23 و 29) پس از نام کلاس و **::** قرار گرفته است، که بعنوان عملگر *تفکیک قلمرو باینری* شناخته می‌شود. این عملگر مبادرت به «گره زدن» هر تابع عضو با تعریف کلاس **GradeBook** (که هم اکنون جدا شده) می‌کند، که توابع عضو کلاس و اعضای داده را اعلان کرده است. در صورتیکه "**GradeBook::**" قبل از نام تابع قرار داده نشود، این توابع توسط کامپایلر بعنوان توابع عضو از کلاس **GradeBook** تشخیص داده نشده و کامپایلر آنها را همانند توابع «*آزاد*» یا «*بی‌قاعده*» همانند **main** در نظر می‌گیرد. چنین توابعی قادر به دسترسی به داده **private** کلاس **GradeBook** یا فراخوانی توابع عضو کلاس نخواهند بود. بنابر این، کامپایلر نمی‌تواند این توابع را کامپایل نماید. برای مثال، خطوط 19 و 25 که به متغیر **courseName** دسترسی پیدا می‌کنند می‌تواند سبب‌ساز خطای کامپایل شوند، چرا که **courseName** بعنوان یک متغیر محلی در هر تابع اعلان نشده است. کامپایلر اطلاعی ندارد که **courseName** بصورت یک عضو داده کلاس **GradeBook** اعلان شده است.

خطای برنامه‌نویسی

به هنگام تعریف توابع عضو کلاس درخارج از آن کلاس، فراموش کردن نام کلاس و عملگر تفکیک قلمرو باینری (::) قبل از اسامی توابع، خطای کامپایل بدنبال خواهد داشت.

برای نشان دادن این که توابع عضو در GradeBook.cpp بخشی از کلاس **GradeBook** هستند، ابتدا فایل سرآیند GradeBook.h را وارد کرده‌ایم (خط 8 از شکل ۱۲-۳). این کار به ما اجازه دسترسی به کلاسی بنام **GradeBook** در فایل GradeBook.cpp را می‌دهد. به هنگام کامپایل GradeBook.cpp، کامپایلر از اطلاعات موجود در **GradeBook** استفاده می‌کند تا مطمئن شود که

۱- اولین خط هر تابع عضو (خطوط 11، 17، 23 و 29) با نمونه اولیه خود در فایل GradeBook.h مطابقت دارد. برای مثال، کامپایلر مطمئن می‌شود که **getCourseName** پارامتری نمی‌پذیرد و یک رشته برگشت می‌دهد.

۲- هر تابع عضو، اعضای داده کلاس و سایر توابع را می‌شناسد. برای مثال، خط 19 و 25 می‌توانند به متغیر **courseName** دسترسی پیدا کنند، چرا که در GradeBook.h بعنوان یک عضو داده اعلان شده است، و خطوط 13 و 32 می‌تواند توابع **setCourseName** و **getCourseName** را به ترتیب فراخوانی نمایند، چرا که هر کدامیک از آنها بعنوان یک تابع عضو کلاس در GradeBook.h اعلان شده‌اند.

```
1  // Fig. 3.12: GradeBook.cpp
2  // GradeBook member-function definitions. This file contains
3  // implementations of the member functions prototyped in GradeBook.h.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  #include "GradeBook.h" // include definition of class GradeBook
9
10 // constructor initializes courseName with string supplied as argument
11 GradeBook::GradeBook( string name )
12 {
13    setCourseName( name ); // call set function to initialize courseName
14 } // end GradeBook constructor
15
16 // function to set the course name
17 void GradeBook::setCourseName( string name )
18 {
19    courseName = name; // store the course name in the object
20 } // end function setCourseName
21
22 // function to get the course name
23 string GradeBook::getCourseName()
24 {
25    return courseName; // return object's courseName
26 } // end function getCourseName
27
28 // display a welcome message to the GradeBook user
29 void GradeBook::displayMessage()
30 {
31    // call getCourseName to get the courseName
32    cout << "Welcome to the grade book for\n" << getCourseName()
33       << "!" << endl;
34 } // end function displayMessage
```

شکل ۱۲-۳ | تعاریف تابع عضو GradeBook نشاندهنده ساختار پیاده‌سازی کلاس GradeBook.

تست کلاس GradeBook

برنامه شکل ۱۳-۳ همان کار دستکاری شی GradeBook بکار رفته در برنامه ۱۰-۳ را انجام می‌دهد. جداسازی واسط GradeBook از بخش پیاده‌سازی توابع عضو تاثیری در روش استفاده این کد

سرویس‌گیرنده از کلاس ندارد و فقط بر نحوه کامپایل برنامه و لینک آن تاثیر دارد که در مورد آن صحبت خواهیم کرد.

```
1  // Fig. 3.13: fig03_13.cpp
2  // GradeBook class demonstration after separating
3  // its interface from its implementation.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  #include "GradeBook.h" // include definition of class GradeBook
9
10 // function main begins program execution
11 int main()
12 {
13    // create two GradeBook objects
14    GradeBook gradeBook1( "CS101 Introduction to C++ Programming" );
15    GradeBook gradeBook2( "CS102 Data Structures in C++" );
16
17    // display initial value of courseName for each GradeBook
18    cout<<"gradeBook1 created for course:"<< gradeBook1.getCourseName()
19       <<"\ngradeBook2 created for course:"<< gradeBook2.getCourseName()
20       << endl;
21    return 0; // indicate successful termination
22 } // end main
```

```
gradeBook1 created for course : CS101 Introduction to C++ Programming
gradeBook2 created for course : CS102 Data Structures in C++
```

شکل ۱۳-۳ | کلاس GradeBook پس از جداسازی واسط از پیاده‌سازی.

همانند برنامه شکل ۱۰-۳، خط 8 برنامه شکل ۱۳-۳ شامل فایل سرآیند GradeBook.h است و از اینرو است که کامپایلر می‌تواند از ایجاد و دستکاری صحیح شی‌های **GradeBook** در کد سرویس‌گیرنده مطمئن گردد. قبل از اجرای این برنامه، باید فایل‌های کد منبع در شکل ۱۲-۳ و ۱۳-۳ هر دو کامپایل‌شده و سپس به هم لینک گردند. فراخوانی تابع عضو در کد سرویس‌گیرنده نیازمند گره خوردن با پیاده‌سازی توابع عضو کلاس دارد، کاری که لینکر آن را انجام می‌دهد.

فرآیند کامپایل و لینک

دیاگرام شکل ۱۴-۳ نمایشی از فرآیند کامپایل و لینک است که نتیجه آن یک برنامه اجرایی **GradeBook** بوده که می‌تواند توسط استاد بکار گرفته شود. غالباً واسط کلاس و ساختار پیاده‌سازی توسط یک برنامه‌نویس ایجاد و کامپایل می‌شود و توسط برنامه‌نویس دیگری که کد سرویس‌گیرنده کلاس را پیاده‌سازی کرده است، بکار گرفته می‌شود. بنابر این دیاگرام نشان‌دهنده نیازهای هر دو طرف برانه‌نویس کلاس و برنامه‌نویس کد سرویس‌گیرنده است. خطوط خط‌چین در دیاگرام نشاندهنده قسمت‌های مورد نیاز برنامه‌نویس کلاس، برنامه‌نویس کد سرویس‌گیرنده و کاربر برنامه **GradeBook** هستند. [نکته: شکل ۱۴-۳ یک دیاگرام UML نیست.]

برنامه‌نویس کلاس مسئول ایجاد یک کلاس **GradeBook** با قابلیت استفاده مجدد در ایجاد فایل سرآیند GradeBook.h و فایل کد منبع GradeBook.cpp است که فایل سرآیند را وارد برنامه کرده (**#include**)، سپس فایل کد منبع را برای ایجاد کد شی **GradeBook** کامپایل می‌کند. برای پنهان ساختن جزئیات پیاده‌سازی توابع عضو **GradeBook**، برنامه‌نویس کلاس مبادرت به تدارک دیدن فایل سرآیند GradeBook.h برای برنامه‌نویس کد سرویس‌گیرنده و کد شی برای کلاس **GradeBook** می‌کند که حاوی دستورالعمل‌های زبان ماشین است که نشاندهنده توابع عضو می‌باشد. برنامه‌نویس کد سرویس‌گیرنده، فایل کد منبع را بطور آشکار بدست نمی‌آورد، از اینرو سرویس‌گیرنده از نحوه پیاده‌سازی توابع عضو **GradeBook** بی‌اطلاع باقی خواهد ماند.

شکل ۱۴-۳ | فرآیند کامپایل و لینک که یک برنامه اجرایی تولید می‌کند.

کد سرویس‌گیرنده فقط نیاز به شناخت واسط **GradeBook** به منظور نحوه استفاده از کلاس داشته و بایستی قادر به لینک آن به کد شی خود باشد. از آنجا که واسط کلاس بخشی از تعریف کلاس در فایل سرآیند GradeBook.h است، باید برنامه‌نویس کد سرویس‌گیرنده به این فایل دسترسی داشته و آن را در فایل کد منبع سرویس‌گیرنده وارد سازد (**#include**). زمانیکه کد سرویس‌گیرنده کامپایل می‌شود، کامپایلر از تعریف کلاس در GradeBook.h برای اطمینان از اینکه تابع **main** مبادرت به ایجاد و دستکاری صحیح شی‌های کلاس **GradeBook** می‌کند، استفاده می‌نماید. برای ایجاد برنامه اجرایی **GradeBook** قابل استفاده برای استاد (مربی)، آخرین مرحله لینک بصورت زیر است

۱- کد شی برای تابع **main** (یعنی، کد سرویس‌گیرنده)

۲- کد شی برای کلاس پیاده‌سازی‌کننده تابع عضو کلاس **GradeBook**

۳- کد شی کتابخانه استاندارد C++ برای کلاس‌های C++ (همانند **string**) بکار رفته توسط برنامه‌نویس پیاده‌سازی‌کننده کلاس و برنامه‌نویس کد سرویس‌گیرنده.

خروجی لینکر، برنامه اجرایی GradeBook است که استاد می‌تواند با استفاده از آن نمرات دانشجویان را مدیریت نماید. برای کسب اطلاعات بیشتر در ارتباط با کامپایل برنامه‌های با چند فایل منبع، به مستندات کامپایلر خود مراجعه کنید.

۱۰-۳ اعتبارسنجی داده با توابع set

در بخش ۶-۳ به معرفی توابع set پرداختیم که به سرویس‌گیرنده‌های کلاس اجازه تغییر در مقدار یک عضو داده **private** را می‌دادند. در برنامه شکل ۵-۳، کلاس **GradeBook** مبادرت به تعریف تابع عضو **setCourseName** کرده که فقط مقدار دریافتی از پارامتر **name** خود را به عضو داده **courseName**

تخصیص می‌دهد. این عضو داده مطمئن نیست که نام دورهٔ دریافتی مطابق با یک فرمت مشخص یا معتبر است.

همانطوری‌که در ابتدا بحث مطرح کردیم، فرض می‌کنیم که دانشگاه می‌تواند از برگه ثبت‌نام دانشجو که در آن اسامی دوره فقط 25 کاراکتر یا کمتر طول دارند، چاپ بگیرد. اگر دانشگاه از سیستمی استفاده نماید که حاوی شی‌های **GradeBook** برای تولید رونوشت ثبت‌نامی است، باید کاری کنیم که کلاس **GradeBook** مطمئن شود که عضو داده **courseName** هرگز بیش از 25 کاراکتر نخواهد داشت. برنامه شکل‌های ۱۵-۳ الی ۱۷-۳ سبب افزایش قابلیت تابع عضو **setCourseName** برای انجام فرآیند اعتبارسنجی می‌شوند.

تعریف کلاس GradeBook

توجه کنید که تعریف کلاس **GradeBook** در شکل ۱۵-۳ و واسط آن، یکسان با شکل ۱۱-۳ است. از آنجا که واسط بدون تغییر باقی مانده، سرویس‌گیرنده‌های این کلاس به هنگام تغییر در تعریف تابع **setCourseName**، نیازی به اصلاح نخواهند داشت. این ویژگی سبب می‌شود که سرویس‌گیرنده‌ها از مزیت کلاس ارتقاء یافته **GradeBook** به آسانی و با لینک کد سرویس‌گیرنده به کد شی **GradeBook** ارتقاء یافته، برخوردار شوند.

اعتبارسنجی نام دوره با تابع عضو setCourseName

ارتقاء و بهبود کلاس **GradeBook** در تعریف تابع عضو **setCourseName** صورت می‌گیرد (شکل ۱۶-۳، خطوط 31-18). عبارت **if** در خطوط 21-20 تعیین می‌کند که آیا پارامتر **name** حاوی نام یک دوره معتبر (رشته‌ای به طول 25 کاراکتر یا کمتر) است یا خیر.

اگر نام دوره معتبر باشد، خط 21 مبادرت به ذخیره نام دوره در عضو داده **courseName** می‌کند. به عبارت **name.length()** در خط 20 توجه کنید. این عبارت فراخوانی یک تابع عضو همانند **myGradeBook.displayMessage()** است. کلاس **string** متعلق به کتابخانه استاندارد C++ دارای تابع عضوی بنام **length** است که تعداد کاراکترهای موجود در یک شی **string** را برگشت می‌دهد. پارامتر **name** یک شی از نوع رشته (*string*) است و از اینرو فراخوانی **name.length()** تعداد کاراکترهای موجود در **name** را برگشت می‌دهد. اگر این مقدار کمتر یا برابر 25 باشد، نام دریافتی معتبر بوده و خط 21 اجرا می‌شود.

```
1  // Fig. 3.15: GradeBook.h
2  // GradeBook class definition presents the public interface of
3  // the class. Member-function definitions appear in GradeBook.cpp.
4  #include <string> // program uses C++ standard string class
5  using std::string;
6
```

```
7  // GradeBook class definition
8  class GradeBook
9  {
10 public:
11    GradeBook(string);//constructor that initializes a GradeBook object
12    void setCourseName(string);//function that sets the course name
13   string getCourseName();//function that gets the course name
14    void displayMessage(); //function that displays a welcome message
15 private:
16    string courseName;//course name for this GradeBook
17 }; // end class GradeBook
```

شکل ۱۵-۳ | تعریف کلاس GradeBook.

```
1  // Fig. 3.16: GradeBook.cpp
2  // Implementations of the GradeBook member-function definitions.
3  // The setCourseName function performs validation.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  #include "GradeBook.h" // include definition of class GradeBook
9
10 // constructor initializes courseName with string supplied as argument
11 GradeBook::GradeBook( string name )
12 {
13    setCourseName( name ); // validate and store courseName
14 } // end GradeBook constructor
15
16 // function that sets the course name;
17 // ensures that the course name has at most 25 characters
18 void GradeBook::setCourseName( string name )
19 {
20    if ( name.length() <= 25 ) // if name has 25 or fewer characters
21       courseName = name; // store the course name in the object
22
23    if ( name.length() > 25 ) // if name has more than 25 characters
24    {
25       // set courseName to first 25 characters of parameter name
26       courseName = name.substr( 0, 25 ); // start at 0, length of 25
27
28       cout << "Name \"" << name << "\" exceeds maximum length (25).\n"
29          << "Limiting courseName to first 25 characters.\n" << endl;
30    } // end if
31 } // end function setCourseName
32
33 // function to get the course name
34 string GradeBook::getCourseName()
35 {
36    return courseName; // return object's courseName
37 } // end function getCourseName
38
39 // display a welcome message to the GradeBook user
40 void GradeBook::displayMessage()
41 {
42    // call getCourseName to get the courseName
43    cout << "Welcome to the grade book for\n" << getCourseName()
44       << "!" << endl;
45 } // end function displayMessage
```

شکل ۱۶-۳ | تعریف تابع عضو برای کلاس GradeBook با تابع set که مبادرت به اعتبارسنجی طول عضو داده courseName می‌کند.

عبارت **if** در خطوط 30-23 به حالتی رسیدگی می‌کند که **setCourseName** نام یک دورهٔ نامعتبر دریافت کرده است (نامی که بیش از 25 کاراکتر طول دارد). حتی اگر پارامتر **name** بسیار طولانی باشد، می‌خواهیم که شی **GradeBook** را در یک وضعیت پایدار حفظ کنیم. وضعیتی که در آن عضو داده **courseName** حاوی یک مقدار معتبر باشد (رشته‌ای بطول 25 کاراکتر یا کمتر). از اینرو، مبادرت به کوتاه کردن نام دوره و تخصیص 25 کاراکتر اول **name** به عضو داده **courseName** می‌کنیم (البته این روش کوتاه‌سازی نام دوره چندان جالب نیست). کلاس استاندارد **string** دارای تابع عضوی بنام **substr** (کوتاه شده جمله "*substring*") است که یک شی جدید **string** با کپی کردن بخشی از شی **string** موجود، تهیه می‌کند. با فراخوانی خط 26، عبارت **name.substr(0,25)** دو عدد صحیح (0, 25) به تابع عضو **substr** شی **name** ارسال می‌شوند. این آرگومان‌ها نشاندهنده بخشی از رشته **name** هستند که **substr** آن را برگشت خواهد داد. آرگومان اول نشان‌دهنده موقعیت شروع در رشته اصلی است که کاراکترها از آن موقعیت شروع به کپی شدن خواهند کرد، توجه کنید که موقعیت اولین کاراکتر در هر رشته‌ای با صفر شروع می‌شود. آرگومان دوم نشان‌دهنده تعداد کاراکترهایی است که باید کپی شوند. بنابر این با فراخوانی خط 26، بیست و پنج کاراکتر از رشته **name** از موقعیت صفر برگشت داده خواهد شد. برای مثال اگر نام موجود نگهداری شده "CS101 Introduction to Programming in C++" باشد، تابع **substr** رشته "CS101 Introduction to Pro" را برگشت خواهد داد. پس از فراخوانی **substr**، خط 26 زیر رشته برگشتی توسط **substr** را به عضو داده **courseName** تخصیص می‌دهد. در این حالت، تابع عضو **setCourseName** مطمئن خواهد بود که **courseName** همیشه یک رشته بطول 25 کاراکتر یا کمتر خواهد داشت. اگر تابع عضو مجبور به کوتاه کردن نام دوره برای تبدیل آن به یک مقدار معتبر شود، خطوط 29-28 یک پیغام هشدار به نمایش در می‌آورند.

توجه کنید عبارت **if** در خطوط 30-23 حاوی دو عبارت در بدنه خود است، یکی برای تنظیم **courseName** با 25 کاراکتر اول از پارامتر **name** و یکی برای چاپ پیغام اطلاع دهنده به کاربر. مایل بودیم تا هر دو این عبارات در زمانیکه طول name طولانی‌تر از حد مجاز باشند اجرا گردند، بنابر این هر دو آنها را در درون یک جفت براکت، **{}** قرار داده‌ایم. از فصل دوم بخاطر دارید که این براکت‌ها بلوک ایجاد می‌کنند. در فصل چهارم اطلاعات بیشتری در زمینه قرار دادن عبارات مضاعف در بدنه یک عبارت کنترلی بدست خواهید آورد.

دقت کنید که عبارت cout در خطوط 29-28 بدون عملگر درج در ابتدای خط دوم ظاهر شده است:

```
cout<<"Name\""<<name<<"\"exceeds maximum length(25).\n"
   "Limiting courseName to first 25 characters.\n"<<endl;
```

کامپایلر C++ مبادرت به ترکیب رشته‌های لیترال مجاور هم می‌کند، حتی اگر این رشته‌ها بر روی خطوط مجزا شده از هم در برنامه قرار داشته باشند. بنابر این در عبارت فوق، کامپایلر C++ شروع به ترکیب رشته‌های لیترال "\"exceeds maximum length(25).\n" و "Limiting courseName to first 25 characters.\n" در یک رشته لیترال واحد در خروجی همانند خطوط 29-28 برنامه شکل ۱۶-۳ می‌کند. چنین رفتاری امکان می‌دهد تا رشته‌های طولانی را در چندین خط قرار دهید بدون اینکه مجبور به استفاده از چندین عملگر درج باشید.

```
1  // Fig. 3.17: fig03_16.cpp
2  // Create and manipulate a GradeBook object; illustrate validation.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "GradeBook.h" // include definition of class GradeBook
8
9  // function main begins program execution
10 int main()
11 {
12    // create two GradeBook objects;
13    // initial course name of gradeBook1 is too long
14    GradeBook gradeBook1( "CS101 Introduction to Programming in C++" );
15    GradeBook gradeBook2( "CS102 C++ Data Structures" );
16
17    // display each GradeBook's courseName
18    cout << "gradeBook1's initial course name is: "
19       << gradeBook1.getCourseName()
20       << "\ngradeBook2's initial course name is: "
21       << gradeBook2.getCourseName() << endl;
22
23    // modify myGradeBook's courseName (with a valid-length string)
24    gradeBook1.setCourseName( "CS101 C++ Programming" );
25
26    // display each GradeBook's courseName
27    cout << "\ngradeBook1's course name is: "
28       << gradeBook1.getCourseName()
29       << "\ngradeBook2's course name is: "
30       << gradeBook2.getCourseName() << endl;
31    return 0; // indicate successful termination
32 } // end main
```

```
Name "CS101 Introduction to Programming in C++" exceeds maximum length
(25).
Limiting courseName to first 25 characters.

gradeBook1's initial course name is : CS101 C++ Introduction to pro
gradeBook2's initial course name is : CS102 C++ Data Structures

gradeBook1's course name is : CS101 C++ Programing
```

```
gradeBook2's course name is : CS102 C++ Data Structures
```

شکل ۱۷-۳ | ایجاد و دستکاری شی GradeBook که در آن نام دوره محدود به 25 کاراکتر است.

تست کلاس GradeBook

برنامه شکل ۱۷-۳ به توصیف نسخه اصلاح شده کلاس **GradeBook** (برنامه‌های ۱۵-۳ و ۱۶-۳) در زمینه اعتبارسنجی است. در خط 14 یک شی **GradeBook** بنام **gradeBook1** ایجاد شده است. بخاطر دارید که سازنده **GradeBook** تابع عضو **setCourseName** را برای مقداردهی اولیه عضو داده **courseName** فراخوانی می‌کرد. در نسخه‌های قبلی این کلاس، مزیت فراخوانی **setCourseName** توس‌ط سازنده مورد بررسی قرار نگرفت، که در اینجا به بررسی آن می‌پردازیم. سازنده از مزیت اعتبارسنجی تدارک دیده شده توسط **setCourseName** برخوردار می‌شود. سازنده بجای اینکه کد اعتبارسنجی خود را تکثیر یا تکرار کند، فقط مبادرت به فراخوانی ساده **setCourseName** می‌کند. زمانیکه خط 14 از برنامه شکل ۷-۳ نام اولیه دورهٔ "CS101 Introduction to Programming in C++" را به سازنده **GradeBook** ارسال می‌کند، سازنده این مقدار را به **setCourseName** انتقال می‌دهد، جائیکه مقداردهی اولیه واقعی در آنجا اتفاق می‌افتد. به دلیل اینکه نام دورهٔ حاوی بیش از 25 کاراکتر است، بدنه دومین عبارت **if** اجرا می‌شود، و در نتیجه **courseName** با 25 کاراکتر اول نام دوره یعنی "CS101 Introduction to Pro" مقداردهی می‌گردد. توجه کنید که خروجی در شکل ۱۷-۳ حاوی پیغام هشدار ایجاد شده توسط خطوط 29-28 از شکل ۱۶-۳ در تابع عضو **setCourseName** است. خط 15 یک شی دیگر از **GradeBook** بنام **gradeBook2** ایجاد می‌کند. نام یک دوره معتبر که دقیقا برابر 25 کاراکتر است به سازنده ارسال شده است.

خطوط 21-18 از شکل ۱۷-۳ نام دوره کوتاه شده برای **gradeBook1** و نام دوره برای **gradeBook2** را به نمایش در می‌آورند. خط 24 مستقیماً تابع عضو **setCourseName** شی **gradeBook1** را فراخوانی می‌کند تا نام دوره در شی **GradeBook** را به یک نام کوتاهتر تغییر دهد تا دیگر نیازی به کوتاه‌سازی آن نباشد سپس خطوط 30-27 اسامی دوره را مجدداً به نمایش در می‌آورند.

نکاتی دیگر در ارتباط توابع set

یک تابع *set* **public** همانند **setCourseName** بایستی بدقت مراقب هرگونه تغییر در مقدار یک عضو داده (همانند **courseName**) باشد تا مطمئن گردد که مقدار جدید مناسب این ایتم داده است. برای مثال، مبادرت به تنظیم روزی از ماه به 37 نبایستی قبول شود، مبادرت به تنظیم وزن یک شخص با صفر یا مقدار منفی برای آن نبایستی پذیرفته شود یا در صورتیکه حداکثر امتیاز یا نمره شخصی می‌تواند بین صفر تا 100 باشد، نبایستی امتیاز 185 تایید شود.


مهندسی نرم‌افزار

*با ایجاد عضوهای دادهٔ **private** و کنترل دسترسی به آنها بویژه دسترسی نوشتاری مناسب از طریق توابع عضو **public** می‌تواند در حفظ جامعیت داده کمک‌کننده باشد.*


اجتناب از خطا

*حفظ جامعیت داده بخودی خود بوجود نمی‌آید چرا که اعضای داده بصورت **private** ایجاد می‌شوند و از اینرو بایستی برنامه‌نویس اعتبارسنجی‌های دقیق را انجام داده و خطاها را گزارش نماید.*


مهندسی نرم‌افزار

*توابع عضوی که مبادرت به تنظیم مقادیر داده **private** می‌کنند بایستی مراقب صحیح بودن مقادیر جدید باشند، در صورتیکه این مقادیر معتبر نباشند، باید این توابع تنظیم‌کننده، سعی نمایند تا اعضای داده **private** در وضعیت مناسبی قرار گیرند.*

توابع set یک کلاس می‌توانند مقادیری به سرویس‌گیرنده‌های کلاس برگشت دهند تا نشان دهند که مبادرت به تخصیص یک داده نامعتبر به شی از کلاس شده است. سرویس‌گیرنده کلاس می‌تواند مبادرت به تست مقدار برگشتی از یک تابع set کند تا تعیین نماید که آیا عملیات اصلاح یا تغییر در شی موفقیت‌آمیز بوده است یا خیر و در هر دو حالت کار مقتضی را انجام دهد. در فصل ۱۶ به بررسی نحوه اطلاع‌دهی؛ سرویس‌گیرنده‌های کلاس از طریق مکانیزم رسیدگی به استثناء خواهیم پرداخت. برای اینکه برنامه ۳-۱۵ الی ۳-۱۷ را فعلاً در یک سطح ساده نگهداری کنیم، تابع **setCourseName** در برنامه ۳-۱۶ فقط مبادرت به چاپ پیغام مقتضی در صفحه نمایش می‌کند.

۳-۱۱ مبحث آموزشی مهندسی نرم‌افزار: شناسایی کلاس‌های موجود در مستند نیازهای ATM

در این بخش شروع به طراحی سیستم ATM می‌کنیم که در فصل دوم به معرفی آن پرداختیم. در این بخش به شناسایی کلاس‌ها می‌پردازیم که در ایجاد سیستم ATM مورد نیاز هستند و این کار را با تحلیل اسامی و اصطلاحات آمده در مستند نیازهای ATM می‌کنیم. همچنین به معرفی دیاگرام‌های کلاس UML به منظور مدل کردن روابط مابین این کلاس‌ها خواهیم پرداخت. این کار بعنوان اولین گام در تعریف ساختار سیستم از اهمیت خاصی برخوردار است.

شناسایی کلاس‌ها در سیستم

فرآیند OOD خود را با شناسایی کلاس‌های مورد نیاز در ایجاد یک سیستم ATM آغاز می‌کنیم. سرانجام این کلاس‌ها را با استفاده از دیاگرام‌های کلاس UML و پیاده‌سازی این کلاس‌ها در C++ توصیف خواهیم کرد. ابتدا، نگاهی به مستند نیازها در بخش ۸-۲ می‌اندازیم تا اسامی و اصطلاحات کلیدی را که می‌توانند به ما در شناسایی کلاس‌های که در ایجاد سیستم ATM نقش دارند، پیدا کنیم.

امکان دارد تصمیم بگیریم که برخی از این اسامی و اصطلاحات جزو صفات کلاس‌های دیگر در سیستم باشند. همچنین می‌توانیم استنتاج کنیم که برخی از اسامی ارتباطی با بخش‌های سیستم ندارند و نیازی نیست که آنها را مدل‌سازی نمائیم. کلاس‌های اضافی می‌توانند میزان حرکت ما را در فرآیند طراحی آشکار کنند.

جدول شکل ۱۸-۳ لیستی از اسامی و اصطلاحات موجود در مستند نیازها را به نمایش در آورده است. در این لیست ترتیب از سمت چپ به راست و با توجه به ظاهر شدن این اسامی در مستند نیازها است.

اسامی و اصطلاحات موجود در مستند نیازها		
بانک	پول/ سرمایه	شماره حساب
ATM	صفحه نمایش	PIN
کاربر	صفحه کلید	پایگاه داده بانک
مشتری	تحویل‌دار خودکار	درخواست موجودی
تراکنش	اسکناس 20 دلاری/ نقد	برداشت پول
حساب	شکاف سپرده‌گذاری	سپرده
موجودی	پاکت سپرده	

شکل ۱۸-۳ | اسامی و اصطلاحات موجود در مستند نیازها.

کلاس‌ها را فقط برای اسامی و اصطلاحاتی که در سیستم ATM از اهمیت برخوردار هستند ایجاد می‌کنیم. نیازی به مدل کردن بانک بعنوان یک کلاس نداریم، چرا که بانک بخشی از سیستم ATM نیست، بانک فقط از ما خواسته که سیستم ATM را ایجاد کنیم. مشتری و کاربر نیز نشاندهنده موجودیت‌های خارج از سیستم هستند، این موجودیت‌ها از اهمیت برخوردار می‌باشند چرا که آنها در تعامل با سیستم قرار می‌گیرند، اما نیازی به مدل کردن آنها به عنوان کلاس‌ها در نرم‌افزار ATM نداریم. بخاطر دارید که کاربر ATM (مشتری بانک) را بصورت یک بازیگر در دیاگرام حالت شکل ۱۸-۲ به نمایش در آوردیم.

نیازی به مدل کردن «اسکناس 20 دلاری» یا «پاکت سپرده» بعنوان کلاس نیست. این موارد شی‌های فیزیکی در دنیای واقعی هستند، اما بخشی از فرآیند اتوماتیک‌سازی نمی‌باشند. می‌توانیم بقدر کفایت

اسکناس را در سیستم با استفاده از صفات کلاسی که پرداخت‌کننده یا تحویل‌دار اتوماتیک را مدل می‌کند، عرضه کنیم. در بخش ۱۳-۴ به تخصیص صفات به کلاس‌ها خواهیم پرداخت. برای مثال، تحویل‌دار اتوماتیک مبادرت به نگهداری تعدادی اسکناس در خود می‌کند. مستند نیازها چیزی در ارتباط با کاری که سیستم باید با پاکت سپرده‌ها پس از دریافت آنها انجام دهد، بیان نمی‌کند. می‌توانیم فرض کنیم که فقط تایید وصول پاکت صورت می‌گیرد، عملیاتی که توسط کلاس مدل شده برای شکاف سپرده انجام می‌شود. در بخش ۲۲-۶ با نحوه تخصیص عملیات به کلاس‌ها آشنا خواهید شد.

در این سیستم ATM ساده شده، نمایش مقادیر مختلف "پولی" شامل "موجودی" یک حساب، همانند صفات سایر کلاس‌ها ضروری بنظر می‌رسد. همچنین، اسامی "شماره‌حساب" و "PIN" نشاندهنده اطلاعات با ارزش در سیستم ATM هستند. این موارد از صفات مهم در یک حساب بانکی هستند. با این وجود، نشاندهنده رفتاری نمی‌باشند. از اینرو بهتر خواهد بود تا آنها را بعنوان صفات کلاس حساب مدل‌سازی کنیم.

با وجود آنکه در مستند نیازها کلمه "تراکنش" بصورت کلی بکار گرفته شده است، اما فعلاً قصد مدل کردن این مفهم کلی در تراکنش مالی را نداریم. بجای آن سه نوع تراکنش ("نمایش موجودی"، "برداشت پول" و "سپرده") را بعنوان کلاس‌های مجزا مدل می‌کنیم. این کلاس‌ها دارای صفات خاص مورد نیاز برای انجام تراکنش‌های متعلق بخود را دارا هستند. برای مثال، کلاس برداشت پول نیاز به داشتن میزان پولی دارد که کاربر می‌خواهد برداشت کند. با این وجود، کلاس موجودی، نیازی به داده‌های اضافی ندارد. علاوه بر اینها، سه کلاس تراکنشی رفتارهای منحصر به فردی را در معرض دید قرار می‌دهند. کلاس برداشت پول شامل پرداخت پول به کاربر است، در حالیکه سپرده‌گذاری مستلزم دریافت پاکت سپرده‌گذاری از سوی کاربر است. [نکته: در بخش ۱۰-۱۳، مبادرت به فاکتورگیری ویژگی‌های مشترک از تمام تراکنش‌ها بصورت یک کلاس «تراکنش» کلی خواهیم کرد. با استفاده از مفهوم کلاس‌های انتزاعی و توارث در برنامه‌نویسی شی‌گرا.]

کلاس‌های مورد نیاز سیستم را بر پایه اسامی و اصطلاحات موجود در جدول ۱۸-۳ تعیین می‌کنیم. هر کدامیک از این اسامی به یک یا چند مورد از عبارات زیر مراجعه دارند:

- ATM
- صفحه نمایش
- صفحه کلید

- تحویل‌دار اتوماتیک
- شکاف سپرده‌گذاری
- حساب
- پایگاه داده بانک
- نمایش موجودی (درخواست موجودی)
- برداشت پول
- سپرده‌گذاری

عناصر موجود در این لیست از قابلیت تبدیل شدن به کلاس‌های مورد نیاز در پیاده‌سازی سیستم ATM برخوردار هستند. اکنون می‌توانیم شروع به مدلسازی کلاس‌ها در سیستم خود بر پایه لیست فوق کنیم. اسامی کلاس‌ها را در فرآیند طراحی با حروف بزرگ نشان می‌دهیم (قاعده UML)، همین کار را به هنگام پیاده‌سازی طراحی به زبان C++ انجام خواهیم داد. اگر نام کلاسی بیش از یک کلمه باشد، کلمات را در کنار هم قرار می‌دهیم و هر کلمه را با حرف بزرگ شروع می‌کنیم (مثلاً **MultipleWordName**). با استفاده از این روش، مبادرت به ایجاد کلاس‌های ATM، **Screen** (صفحه نمایش)، صفحه‌کلید (**Keypad**)، تحویل‌دار خودکار یا پرداخت‌کننده پول (**CashDispencer**)، شکاف سپرده (**DepositSlot**)، حساب (**Account**)، پایگاه داده بانک (**BankDatabase**)، پرس‌وجوی میزان موجودی (**BalanceInquiry**)، برداشت پول (**Withdrawal**) و سپره (**Deposit**). سیستم خود را با استفاده از تمام این کلاس‌ها بعنوان بلوک‌های سازنده ایجاد خواهیم کرد. قبل از شروع به ایجاد بلوک‌های سازنده سیستم، بایستی درک مناسبی از روابط مابین کلاس‌ها بدست آوریم.

مدل‌سازی کلاس‌ها

زبان UML امکان مدل‌سازی کلاس‌های موجود در سیستم ATM و روابط داخلی آنها را از طریق دیاگرام‌های کلاس فراهم آورده است. در شکل ۱۹-۳ کلاس **ATM** نشان داده شده است. در UML، هر کلاس بصورت یک مستطیل با سه بخش مدل می‌شود.

بخش فوقانی حاوی نام کلاس در وسط و بصورت توپر (پر رنگ) نوشته می‌شود. بخش میانی حاوی صفحات کلاس است (در بخش ۱۳-۴ و بخش ۱۱-۵ به بررسی صفات خواهیم پرداخت) بخش تحتانی حاوی عملیات کلاس است (در بخش ۲۲-۶ بحث خواهد شد). در شکل ۱۹-۳ بخش میانی و تحتانی خالی هستند، چرا که هنوز به تعیین صفات و عملیات کلاس نپرداخته‌ایم.

دیاگرام‌های کلاس از قابلیت نمایش روابط مابین کلاس‌های سیستم برخوردار هستند. شکل ۲۰-۳ نشاندهنده نحوه رابطه کلاس‌های **ATM** و **Withdrawal** با یکدیگر است. برای سادگی کار، در این لحظه فقط مبادرت به مدل‌سازی این زیر مجموعه می‌کنیم. دیاگرام کامل کلاس را در ادامه این بخش شاهد خواهید بود. دقت کنید که مستطیل‌های نشاندهنده کلاس‌ها در این دیاگرام به زیربخش‌ها تقسیم نمی‌شوند.

در شکل ۲۰-۳، خط مستقیم و یکپارچه دو کلاس را به هم پیوند زده، نشاندهنده یک *وابستگی* است (رابطه مابین کلاس‌ها). اعداد قرار گرفته در انتهای خط مقادیر تعدد هستند که نشان می‌دهند که چند شی از هر کلاس در وابستگی (رابطه) شرکت یا دخالت دارند.

شکل ۱۹-۳ | نمایش کلاس در UML با استفاده از دیاگرام کلاس.

شکل ۲۰-۳ | دیاگرام‌های کلاس در حال نمایش وابستگی مابین کلاس‌ها.

در این مورد، با دنبال کردن خط از یک طرف به طرف دیگر معلوم می‌شود که در هر لحظه، یک شی ATM در رابطه (وابستگی) با صفر یا یک شی **Withdrawal** شرکت دارد. اگر کاربر جاری در حال حاضر تراکنشی انجام ندهد یا تقاضای یک تراکنش از نوع دیگری را نماید، صفر، و در صورتیکه تقاضای برداشت پول (withdrawal) کند، مقدار 1 بکار گرفته می‌شود. زبان UML قادر به مدل کردن انواع تعدد یا کثرت است. در جدول شکل ۲۱-۳ لیستی از انواع تعددها و مفهوم آنها آورده شده است.

نماد	مفهوم
0	هیچ- اصلاً
1	یک
m	مقدار صحیح
0..1	صفر یا یک
m,n	m یا n
m..n	حداقل m، اما نه بیشتر از n
*****	هر مقدار غیرمنفی (صفر یا بیشتر)

0..*	صفر یا بیشتر (همان‌ند *)
1..*	یک یا بیشتر

شکل ۲۱-۳ | انواع تعدد.

وابستگی می‌تواند نام داشته باشد. برای مثال، کلمه **Executes** در بالای خط متصل‌کننده کلاس‌های **ATM** و **Withdrawal** در شکل ۲۰-۳ نشان‌دهنده نام این وابستگی (ارتباط) است. این بخش از دیاگرام بصورت زیر معنی می‌شود، "یک شی از کلاس **ATM** صفر یا یک شی از کلاس **Withdrawal** را به اجرا در می‌آورد." توجه نمایید که اسامی وابستگی، حالت هدایت‌کننده و نشان‌دهنده جهت هستند، همانند جهت فلش توپر، از اینرو جهت تفسیر دیاگرام مهم است، در صورتی که برای مثال تفسیر را از سمت راست به چپ انجام دهیم، جمله‌ای به این مضمون خواهیم داشت "صفر یا یک شی از کلاس **Withdrawal** یک شی از کلاس **ATM** را به اجرا درمی‌آورد."

کلمه **currentTransaction** در کنار و زیرخط وابستگی **Withdrawal** در شکل ۲۰-۳، نام یک *نقش* (*role*) است، که هویت‌دهنده نقشی است که شی **Withdrawal** در رابطه خود با **ATM** بازی می‌کند. نام نقش، هدف و منظوری را به رابطه مابین کلاس‌ها اضافه می‌کند، و اینکار را با شناسایی نقشی که کلاس در بافت رابطه ایفاء می‌کند، انجام می‌دهد. یک کلاس می‌تواند چندین نقش در همان سیستم بازی کند. برای مثال، در یک سیستم پرسنلی دانشگاه یک نفر می‌تواند نقش یک «پورفسور» را به هنگام رابطه داشتن با دانشجویان بازی کند. امکان دارد همان شخص نقش «همکار» در کنار پورفسور دیگر و نقش «مربی» را به هنگام مسابقات دانشجویی بازی کند. در شکل ۲۰-۳، نام نقش **currentTransaction** بر این نکته دلالت دارد که شی **Withdrawal** در اجرای (**Executes**) رابطه با یک شی از کلاس **ATM** دخالت دارد و این کلاس در پردازش تراکنش جاری عمل می‌کند. در سایر محیط‌ها امکان دارد یک شی **Withdrawal** نقش‌های متفاوتی بخود گیرد. زمانیکه مفهوم رابطه در دیاگرام به قدر کافی گویا باشد، از اسامی نقش در دیاگرام‌های کلاس استفاده نمی‌شود.

علاوه بر وجود روابط ساده، وابستگی می‌تواند تصریح‌کننده روابط بسیار پیچیده و مرکب باشد، همانند زمانیکه شی‌های از یک کلاس با شی‌های از کلاس‌های دیگر ترکیب شوند. به سیستم ATM واقعی توجه کنید. سازنده ATM باید چه قسمت‌های را در کنار هم قرار دهد تا ATM قادر به کار باشد؟ مستند نیازها به ما می‌گوید که ATM دستگاهی متشکل از یک صفحه نمایش، یک صفحه‌کلید، یک پرداخت‌کننده یا تحویل‌دار خودکار و یک شکاف سپرده‌گذاری است.

در شکل ۲۲-۳ *لوزی‌های توپر* متصل شده به خطوط وابستگی کلاس **ATM** بر این نکته دلالت دارند که کلاس **ATM** دارای یک رابطه *ترکیبی* با کلاس‌های **Screen**، **Keypad**، **CashDispenser** و DepositSlot است. ترکیب مفهومی از یک رابطه کامل/بخش (قطعه) است. کلاسی که دارای نماد ترکیب (لوزی توپر) در انتها خط وابستگی خود می‌باشد، کامل بوده (در این مورد **ATM**) و کلاس‌های قرار گرفته در آن سوی خطوط وابستگی، بخش یا قطعه می‌باشند، در این مورد کلاس‌های **Screen**، **Keypad**، **CashDispenser** و **DepositSlot**. ترکیب بنمایش درآمده در شکل ۲۲-۳ نشان می‌دهد که یک شی از کلاس **ATM** از یک شی از کلاس **Screen**، یک شی از کلاس **CashDispenser**، یک شی از کلاس **Keypad** و یک شی از کلاس **DepositSlot** تشکیل یافته است. ATM «*دارای*» یک صفحه نمایش، یک صفحه‌کلید، یک تحویل‌دار خودکار و شکاف سپرده است. رابطه «*داشتن*» تعریف‌کننده ترکیب است. در فصل سیزدهم نشان خواهیم داد که رابطه «*است یک*» تعریف‌کننده توارث است.

بر طبق مشخصات UML، رابطه ترکیب دارای خصوصیات زیر است:

۱- فقط یک کلاس در رابطه می‌تواند نشاندهنده رابطه کامل باشد (لوزی می‌تواند فقط در انتهای یک طرف خط وابستگی قرار گرفته باشد). برای مثال خواه صفحه نمایش بخشی از ATM باشد یا ATM بخشی از صفحه نمایش باشد، صفحه نمایش و ATM هر دو نمی‌توانند نشاندهنده رابطه کامل (سراسری) باشند.

۲- قطعات یا بخش‌ها در رابطه ترکیب تا زمانیکه رابطه کامل وجود دارد، وجود خواهند داشت و این رابطه کامل مسئول ایجاد و نابود کردن قطعات متعلق بخود است. برای مثال، اقدام به ایجاد یک ATM مستلزم ساخت قطعات آن است. علاوه بر این، اگر ATM نابود گردد، صفحه نمایش، صفحه‌کلید، پرداخت‌کننده اتوماتیک و شکاف سپرده آن نیز نابود خواهند شد.

۳- یک قطعه می‌تواند فقط در یک زمان متعلق به یک رابطه کامل باشد، اگر چه امکان دارد قطعه‌ای حذف و به رابطه کامل دیگری متصل گردد.

لوزی‌های بکار رفته در دیاگرام کلاس ما بر این نکته دلالت دارند که در رابطه ترکیبی این سه خصیصه وجود دارند. اگر رابطه «داشتن» قادر به برآوردن یکی از چند ضابطه فوق نباشد، UML بر استفاده از لوزی‌های توخالی متصل شده به انتهای خطوط وابستگی تصریح می‌کند تا نشاندهنده *اجتماع* یا *تراکم* باشند. اجتماع نسخه ضعیف‌تر ترکیب است. برای مثال، یک کامپیوتر و مانیتور در رابطه اجتماع قرار دارند، کامپیوتر «دارای» مانیتور است، اما دو قطعه می‌توانند بصورت مستقل وجود داشته باشند و

همان مانیتور می‌تواند در یک زمان به چندین کامپیوتر متصل گردد، از اینرو اینحالت نقض خصیصه دوم و سوم ترکیب است.

شکل ۲۲-۳ | دیاگرام کلاس نشاندهنده رابطه ترکیب.

در شکل ۲۳-۳ نمایشی از دیاگرام کلاس سیستم ATM ارائه شده است. این دیاگرام اکثر کلاس‌هایی که در ابتدای این بخش شناسایی کرده بودیم را به همراه وابستگی مابین آنها را که از مستند نیازها استنتاج کرده‌ایم، مدل کرده است. [نکته: کلاس‌های **BalanceInquiry** و **Deposit** در رابطه مشابهی با کلاس **Withdrawal** شرکت دارند، از اینرو برای حفظ سادگی دیاگرام، آنها را در نظر نگرفته‌ایم. در فصل ۱۳، دیاگرام کلاس را گسترش داده و تمام کلاس‌های موجود در سیستم ATM را وارد آن می‌کنیم.]

شکل ۲۳-۳ نمایشی از مدل گرافیکی ساختار سیستم ATM است. این دیاگرام کلاس حاوی کلاس‌های **BankDatabase** و **Account** و چندین رابطه (وابستگی) است که در شکل‌های ۲۰-۳ یا ۲۲-۳ عرضه نشده بودند. دیاگرام کلاس نشان می‌دهد که کلاس ATM دارای یک رابطه یک به یک با کلاس **BankDatabase** است، یک شی **ATM** مبادرت به تصدیق کاربران در برابر یک شی **BankDatabase** می‌کند. همچنین در شکل ۲۳-۳، مبادرت به مدل کردن این واقعیت کرده‌ایم که پایگاه بانک حاوی اطلاعاتی در ارتباط با حساب‌های متعدد است، یک شی از کلاس **BankDatabase** در یک رابطه ترکیبی با صفر یا چندین شی از کلاس **Account** شرکت دارد. از جدول ۲۱-۳ بخاطر دارید که مقدار تعدد یا کثرت ***..0** در سمت وابستگی **Account** مابین کلاس **BankDatabase** و کلاس **Account** بر این نکته دلالت دارد که صفر یا چندین شی از کلاس **Account** بخشی در رابطه را تشکیل می‌دهند. کلاس **BankDatabase** دارای رابطه یک به چند با کلاس **Account** است. **BankDatabase** مبادرت به ذخیره حساب‌های (*accounts*) متعدد در خود می‌کند. به همین ترتیب، کلاس **Account** دارای رابطه چند به یک با کلاس **BankDatabase** است، چرا که حساب‌های متعدد در پایگاه داده بانک (*bank database*) ذخیره می‌شوند. [نکته: از جدول شکل ۲۱-۳ بخاطر دارید که مقدار * معادل با ***..0** است. برای افزایش وضوح دیاگرام‌های کلاس از نماد ***..0** استفاده کرده‌ایم.]

شکل ۲۳-۳ | دیاگرام کلاس برای مدل کردن سیستم ATM.

همچنین شکل ۲۳-۳ نشان می‌دهد که اگر کاربر مبادرت به برداشت پول کند، «یک شی از کلاس **Withdrawal** به موجودی حساب دسترسی پیدا کرده/آنرا از طریق یک شی از کلاس **BankDatabase** تغییر می‌دهد.» می‌توانیم یک رابطه مستقیم مابین کلاس **Withdrawal** و کلاس **Account** ایجاد کنیم. با این وجود، مستند نیازها شرح می‌دهد که «**ATM** باید با پایگاه داده اطلاعات حساب بانک در تعامل قرار داشته باشد» تا تراکنش‌ها قابل انجام باشند. حساب بانکی حاوی اطلاعات حساس بوده و مهندسان

سیستم بایستی همیشه مراقب امنیت داده افراد به هنگام طراحی سیستم باشند. از اینرو، فقط **BankDatabase** می‌تواند مبادرت به دسترسی و اعمال تغییر مستقیم در یک حساب کند. تمام قسمت‌های دیگر سیستم باید با پایگاه داده در تعامل قرار گیرند تا بتوانند اطلاعاتی بدست آورده یا حساب را به روز نمایند.

همچنین دیاگرام کلاس در شکل ۲۳-۳ مبادرت به مدل کردن رابطه موجود مابین کلاس **Withdrawal** و کلاس‌های **Screen**، **CashDispenser** و **Keypad** می‌کند. تراکنش برداشت پول شامل اعلان پیغامی به کاربر برای تعیین میزان پول برداشتی و دریافت ورودی عددی است. این اعمال به ترتیب مستلزم استفاده از صفحه نمایش و صفحه کلید است. علاوه بر اینها، پرداخت پول نقد به کاربر مستلزم دسترسی به تحویل‌دار خودکار (پرداخت‌کننده پول خودکار) می‌باشد.

کلاس‌های **BalanceInquiry** و **Deposit** که در شکل ۲۳-۳ آورده نشده‌اند، دارای چندین رابطه با کلاس‌های دیگر در سیستم ATM هستند. همانند کلاس **Withdrawl**، هر کدامیک از این کلاس‌ها با کلاس‌های **ATM** و **BankDatabase** دارای رابطه (وابستگی) هستند. یک شی از کلاس **BalanceInquiry** دارای رابطه‌ای با یک شی از کلاس **Screen** برای نمایش میزان موجودی در حساب یک کاربر نیز است. کلاس **Deposit** با کلاس‌های **Screen**، **Keypad** و **DepositSlot** در ارتباط است. همانند برداشت پول، تراکنش سپرده‌گذاری مستلزم استفاده از صفحه‌نمایش و صفحه‌کلید برای نمایش پیغامی به کاربر و دریافت ورودی است. برای دریافت پاکت سپرده، یک شی از کلاس **Deposit** به شکاف سپرده دسترسی پیدا می‌کند.

اکنون کلاس‌های موجود در سیستم ATM خود را شناسایی کرده‌ایم. در بخش ۱۳-۴ به تعیین صفات هر کدامیک از این کلاس‌ها خواهیم پرداخت. در بخش ۱۱-۵ از این صفات برای بررسی نحوه تغییر عملکرد سیستم در زمان استفاده می‌کنیم. در بخش ۲۲-۶ به تعیین عملیاتی که کلاس‌ها در سیستم انجام خواهند داد، می‌پردازیم.

تمرینات خودآزمایی مبحث آموزشی مهندسی نرم‌افزار

۱-۳ فرض کنید کلاسی بنام **Car** داریم که نشان‌دهنده یک اتومبیل است. در مورد قسمت‌های مختلف آن که سازنده باید درکنار هم قرار دهد تا یک اتومبیل کامل ایجاد گردد، فکر کنید. یک دیاگرام کلاس (شبیه به شکل ۲۲-۳) ایجاد کنید که برخی از روابط ترکیبی موجود در کلاس **Car** را مدل‌سازی کند.

۳-۲ فرض کنید کلاسی بنام **File** داریم که نشاندهنده یک مستند الکترونیکی بر روی یک سیستم کامپیوتری منفرد (بدون اتصال به شبکه) است که توسط کلاس **Computer** نشان داده می‌شود، قرار دارد. چه نوع رابطه (وابستگی) مابین کلاس **Computer** و کلاس **File** وجود دارد؟

a) کلاس **Computer** دارای رابطه یک به یک با کلاس **File** است.

b) کلاس **Computer** دارای رابطه چند به یک با کلاس **File** است.

c) کلاس **Computer** دارای رابطه یک به چند با کلاس **File** است.

d) کلاس **Computer** دارای رابطه چند به چند با کلاس **File** است.

۳-۳ تعیین کنید که آیا عبارت زیر صحیح است یا خیر و در صورتیکه اشتباه باشد علت را توضیح دهید: به یک دیاگرام کلاس UML که بخش دوم و سوم آن مدل نشده‌اند گفته می‌شود یک دیاگرام ادغام شده است.

۳-۴ دیاگرام کلاس شکل ۳-۲۳ را برای وارد کردن کلاس **Deposit** بجای کلاس **Withdrawal** تغییر دهید.

پاسخ خودآزمایی مبحث مهندسی نرم‌افزار

۳-۱ [نکته: پاسخ‌های می‌توانند متفاوت باشند.] شکل ۳-۲۴ نشاندهنده دیاگرام کلاسی است که برخی از روابط ترکیبی در کلاس **Car** را عرضه می‌کند.

۳-۲ c. [نکته: در یک کامپیوتر شبکه، این رابطه می‌تواند بصورت چند به چند باشد.]

۳-۳ صحیح.

۳-۴ شکل ۳-۲۵ نشاندهنده یک دیاگرام کلاس برای **ATM** شامل کلاس **Deposit** بجای کلاس **Withdrawal** (همانند شکل ۳-۲۳) است. توجه کنید که **Deposit** به **CashDispenser** دسترسی ندارد، اما به **DepositSlot** دسترسی دارد.

شکل ۳-۲٤ | دیاگرام کلاس نشاندهنده رابطه ترکیبی در کلاس Car.

شکل ۳-۲۵ | دیاگرام کلاس سیستم ATM حاوی کلاس Deposit.

خودآزمایی

۳-۱ جاهای خالی را با عبارت مناسب پر کنید:

a) نقشه ترسیمی یک خانه همانند یک _____ برای یک کلاس است.

b) تعریف هر کلاس حاوی کلمه کلیدی _____ و بدنبال آن نام کلاس است.

c) تعریف کلاس در فایلی با پسوند ـــــــــــ ذخیره می‌گردد.

d) هر پارامتر در سرآیند یک تابع بایستی توسط ـــــــــــ و ـــــــــــ مشخص گردد.

e) زمانیکه هر شی از یک کلاس مبادرت به نگهداری کپی از صفات خود می‌کند، به متغیری که نشاندهنده صفات است، ـــــــــــ گفته می‌شود.

f) کلمه کلیدی **public**، یک ـــــــــــ است.

g) نوع برگشتی ـــــــــــ بر این نکته دلالت دارد که تابع وظیفه خود را انجام داده اما پس از انجام وظیفه خود مقداری برگشت نمی‌دهد.

h) تابع ـــــــــــ از کتابخانه <**string**> مبادرت به قرائت کاراکترها تا رسیدن به کاراکتر خط جدید می‌کند، سپس این کاراکترها را به رشته مشخص‌شده کپی می‌نماید.

i) به هنگام تعریف تابع عضو در خارج از تعریف کلاس، بایستی سرآیند تابع شامل نام کلاس و ـــــــــــ، بدنبال نام تابع برای «گره زدن» تابع عضو به تعریف کلاس باشد.

j) فایل کد منبع و سایر فایل‌های که از یک کلاس استفاده می‌کنند می‌توانند از طریق رهنمود پیش‌پردازنده ـــــــــــ سرآیند فایل کلاس را شامل گردند.

۳-۲ تعیین کنید کدامیک از عبارات زیر صحیح و کدامیک اشتباه است.

a) بطور قراردادی، اسامی تابع با یک حرف بزرگ و تمام کلمات متعاقب آن در نام با حرف بزرگ شروع می‌شوند.

b) پرانتزهای خالی پس از نام تابع در یک نمونه اولیه تابع نشان می‌دهند که تابع به هیچ پارامتری برای انجام وظیفه خود نیاز ندارد.

c) اعضای داده یا توابع عضو اعلان شده با تصریح‌کننده دسترسی **private** در دسترسی توابع عضو کلاسی قرار دارند که در آن اعلان شده‌اند.

d) متغیرهای اعلان شده در بدنه یک تابع عضو خاص بعنوان اعضای داده شناخته می‌شوند و می‌توانند در تمام توابع عضو کلاس بکار گرفته شوند.

e) بدنه هر تابع توسط یک براکت چپ و راست ({ و }) تعیین می‌شود.

f) هر فایل کد منبع که حاوی **int main()** است می‌تواند در اجرای برنامه بکار گرفته شود.

g) نوع آرگومان‌های موجود در یک تابع فراخوانی شده بایستی با نوع پارامترهای متناظر در لیست پارامتری نوع اولیه تابع مطابقت داشته باشد.

۳-۳ تفاوت موجود مابین یک متغیر محلی و عضو داده در چیست؟

۳-٤ هدف از پارامتر تابع چیست؟ تفاوت موجود مابین یک پارامتر و آرگومان را توضیح دهید.

پاسخ خودآزمایی

۳-۱ a) شی b) **class** c) .h d) نوع، نام. e) عضو داده. f) تصریح‌کننده دسترسی. g) **void** h) **getline** i)عملگر تفکیک قلمرو باینری (::) j) **#include**

۳-۲ a) اشتباه. بطور قراردادی، اسامی توابع با حرف کوچک شروع و تمام کلمات متعاقب آن در نام با حرف بزرگ آغاز می‌شوند. b) صحیح. c) صحیح. d) اشتباه. چنین متغیرهای، متغیرهای محلی نامیده می‌شوند و می‌توانند فقط در تابع عضوی که در آن اعلام شده‌اند بکار گرفته شوند. e) صحیح f) صحیح. g) صحیح.

۳-۳ متغیر محلی در بدنه یک تابع اعلان می‌شود و می‌تواند فقط از نقطه‌ای که تعریف شده تا رسیدن به براکت خاتمه بکار گرفته شود. عضو داده در تعریف کلاس اعلان می‌شود اما در بدنه توابع عضو کلاس قرار ندارد. هر شی (نمونه) یک کلاس دارای یک کپی متمایز از اعضای داده کلاس است. همچنین اعضای داده برای تمام عضو کلاس در دسترس هستند.

۳-٤ پارامتر نشاندهنده اطلاعات اضافی است که تابع برای انجام وظیفه خود به آن نیاز دارد. هر پارامتر مورد نیاز تابع در سرآیند تابع جای داده می‌شود. آرگومان مقداری است که در فراخوانی تابع تدارک دیده می‌شود. زمانیکه تابع فراخوانی می‌شود، مقدار آرگومان به پارامتر تابع ارسال می‌شود، از اینروست که تابع می‌تواند وظیفه خود را انجام دهد.

تمرینات

۳-۵ تفاوت موجود مابین نمونه اولیه تابع و تعریف تابع را بیان کنید.

۳-٦ سازنده پیش‌فرض چیست؟ اگر کلاسی دارای فقط یک سازنده پیش‌فرض ضمنی باشد، اعضای داده شی چگونه مقداردهی اولیه خواهند شد؟

۳-۷ منظور از عضو داده چیست؟

۳-۸ سرآیند فایل چیست؟ فایل کد منبع چیست؟ هدف از هر یک را توضیح دهید.

۳-۹ توضیح دهید چگونه برنامه از کلاس **string** بدون اعلان **using** استفاده می‌کند.

۳-۱۰ توضیح دهید چرا کلاسی مبادرت به تدارک دیدن یک تابع **set** و **get** برای کار با عضو داده می‌کند.

۳-۱۱ (اصلاح کلاس **GradeBook**). کلاس **GradeBook** در شکل‌های ۳-۱۱ و ۳-۱۲ را بصورت زیر تغییر دهید یا اصلاح کنید:

a) یک عضو داده رشته‌ای دیگر اضافه کنید که نشاندهنده نام استاد دوره باشد.

b) یک تابع **set** برای تغییر دادن نام استاد و یک تابع **get** برای بازیابی آن در نظر بگیرید.

c) سازنده را با دو پارامتر، یکی برای نام دورهٔ و دیگری برای نام استاد، تغییر دهید.

d) تابع عضو **displayMessage** را به نحوی تغییر دهید که ابتدا پیغام خوش‌آمدگویی و نام دوره را چاپ کرده و سپس جمله "This course is presented by:" را چاپ و بدنبال آن نام استاد را به نمایش در آورد. کلاس اصلاح شده خود را در برنامه تست بکار گیرید تا قابلیت‌های جدید کلاس عرضه گردد.

۳-۱۲ (کلاس **Account**) کلاسی بنام **Account** ایجاد کنید که در بانک از آن برای نمایش حساب بانکی مشتری استفاده شود. این کلاس بایستی شامل یک داده عضو از نوع **int** برای نمایش میزان موجودی حساب باشد. [نکته: در

فصل‌های بعدی از اعداد اعشاری استفاده خواهیم کرد.] این کلاس باید یک سازنده داشته باشد که موجودی اولیه را دریافت و با استفاده از آن مبادرت به مقداردهی اولیه عضو داده نماید. همچنین سازنده باید میزان موجودی اولیه را اعتبارسنجی کند و مطمئن گردد که این مقدار بزرگتر یا برابر صفر است. اگر چنین نباشد، موجودی را با صفر تنظیم کند و پیغام خطا را به نمایش در آورده تا نشان دهد که موجودی اولیه معتبر نیست. همچنین کلاس باید سه تابع عضو تدارک دیده باشد. تابع عضو **credit** باید مقداری پول به موجودی جاری اضافه کند. تابع عضو **debit** باید از حساب **(Account)** برداشت کند و مطمئن باشد که میزان برداشتی از میزان موجودی حساب تجاوز نکند. اگر چنین باشد، باید موجودی دست نخورده باقی بماند و تابع پیغامی مبنی بر اینکه «میزان درخواستی بیش از میزان موجودی است» موضوع را اطلاع دهد. تابع **getBalance** باید میزان موجودی جاری را برگشت دهد. برنامه‌ای ایجاد کنید که دو شی **Account** ایجاد کرده و توابع عضو کلاس **Account** را تست کنید.

۱۳-۳ (کلاس **Invoice**) کلاسی بنام **Invocie** (فاکتور) ایجاد کنید که در یک فروشگاه سخت‌افزار با هدف نمایش فاکتور از ایتم‌های فروخته شده بکار گرفته شود. این فاکتور باید شامل چهار قسمت اطلاعاتی بعنوان اعضای داده باشد، شماره قطعه (از نوع رشته)، توضیح قطعه (از نوع رشته)، تعداد قطعه فروخته شده (از نوع **int**) و قیمت هر قطعه (از نوع **int**). این کلاس باید دارای سازنده‌ای باشد که مبادرت به مقداردهی اولیه چهار عضو داده کند. یک تابع **set** و **get** برای هر عضو داده در نظر بگیرید. علاوه بر این، یک تابع عضو بنام **getInvoiceAmount** تدارک ببینید که مبادرت به محاسبه فاکتور (با ضرب تعداد در قیمت هر قطعه) کرده و سپس قیمت فاکتور را بصورت یک مقدار **int** برگشت دهد. اگر تعداد، مقدار مثبتی نباشد، باید آنرا با صفر تنظیم کند. اگر قیمت قطعه‌ای مثبت نباشد، آن را با صفر تنظیم کند. یک برنامه تست‌کننده برای این کلاس بنویسید.

۱٤-۳ (کلاس **Employee**) کلاسی بنام **Employee** ایجاد کنید که شامل سه قسمت اطلاعاتی بعنوان اعضای داده باشد، نام (از نوع رشته)، نام خانوادگی (از نوع رشته) و حقوق ماهانه (از نوع **int**). این کلاس باید دارای سازنده‌ای باشد که مبادرت به مقداردهی اولیه سه عضو داده کند. یک برنامه تست برای بررسی قابلیت کلاس **Employee** بنویسید. دو شی **Employee** ایجاد کرده و نمایش حقوق سالیانه هر شی را به نمایش در آورید. سپس افزایش حقوق ده درصدی را برای هر کارمند در نظر گرفته و مجدداً حقوق سالیانه را برای هر کارمند محاسبه و به نمایش در آورید.

۱۵-۳ (کلاس **Date**) کلاسی به نام **Date** ایجاد کنید که شامل سه قسمت اطلاعاتی بعنوان اعضای داده باشد، ماه (از نوع **int**)، روز (از نوع **int**) و سال (از نوع **int**). این کلاس باید دارای یک سازنده با سه پارامتر باشد که از پارامترها برای مقداردهی اولیه این سه عضو داده استفاده می‌کند. در این تمرین فرض کنید که مقادیر تدارک دیده شده برای سال و روز صحیح باشند، اما مطمئن گردید که مقدار ماه حتماً در محدودهٔ 1 الی 12 قرار داشته باشد. اگر چنین نباشد، ماه را با 1 مقداردهی یا تنظیم کنید. برای هر عضو داده یک تابع **set** و **get** در نظر بگیرید. یک تابع **displayDate** بنویسید که ماه، روز و سال را که با یک اسلش (/) از هم جدا شده‌اند به نمایش درآورد. یک برنامه تست برای نمایش قابلیت‌های کلاس **Date** ایجاد کنید.

فصل چهارم

عبارات کنترلی: بخش ۱

اهداف

- آشنائی با تکنیک‌های اصول حل مسائل.
- توسعه الگوریتم‌ها به طریق فرآیندهای از بالا به پایین و اصلاح گام به گام.
- بکارگیری عبارات انتخاب if و if..else برای انتخاب از میان چندین عملکرد مختلف.
- بکارگیری عبارات تکرار while برای اجرای تکراری عبارات در برنامه.
- آشنائی با شمارنده-کنترل تکرار و مراقبت-کنترل تکرار.
- بکارگیری عملگرهای افزایشی، کاهشی و تخصیصی.

رئوس مطالب

٤-١ مقدمه

قبل از نوشتن یک برنامه که بتواند مسئله‌ای را به طور دقیق حل کند، ضروری است که درک صحیحی از مسئله داشته باشیم و بوسیله یک طراحی دقیق به آن اقدام کنیم. به هنگام نوشتن یک برنامه، سازماندهی انواع بلوک‌های ایجاد کننده برنامه به منظور بهبود قوانین عملی ساخت برنامه بسیار مهم است. در این فصل و فصل بعدی در مورد تئوری و قواعد علمی برنامه‌نویسی ساخت‌یافته بحث خواهیم کرد. تکنیک‌های معرفی شده در این فصل در اکثر زبان‌های سطح بالا نظیر کاربرد دارند. با معرفی مفاهیم مطرح شده در اینجا متوجه مزیت عبارتهای کنترل در ایجاد و دستکاری شی‌ها خواهید شد. عبارتهای کنترل معرفی شده در این فصل در ایجاد سریع و آسانتر شی‌ها کمک کننده هستند.

در این فصل به معرفی عبارات **if**، **if..else** و **while** در زبان C++ خواهیم پرداخت. سه بلوک سازنده که به برنامه‌نویسان امکان می‌دهند تا منطق مورد نیاز توابع عضو را برای انجام وظایف مشخص کنند. بخشی از این فصل و فصل‌های ۵ و ۷ را برای توسعه دادن کلاس **GradeBook** معرفی شده در فصل سوم اختصاص داده‌ایم. در واقع، یک تابع عضو به کلاس **GradeBook** اضافه می‌کنیم که از عبارات کنترلی برای محاسبه میانگین نمرات دانشجویان استفاده می‌کند. مثال دیگر به معرفی روش‌های ترکیب عبارات کنترلی برای حل مسئله مشابه است. همچنین به معرفی عملگرهای تخصیص‌دهنده و عملگرهای افزاینده و کاهنده در C++ خواهیم پرداخت. این عملگرها سبب کوتاه شدن عبارات و گاهاً سادگی کار می‌شوند.

٤-٢ الگوریتم

هر مسئله محاسباتی و کامپیوتری می‌تواند با یک سری از اعمال اجرائی که به ترتیب اجرا می‌شوند، حل شود. از روال‌ها برای حل مسائل کمک گرفته می‌شود و عبارات:

۱- فعالیت‌ها از نوع اجرای هستند و

۲- فعالیت‌ها به ترتیب اجرا می‌شوند،

تعریف الگوریتم می‌باشند. با ذکر مثالی که در زیر آمده می‌توانید ترتیب اجرا و فعالیت‌ها را ببینید و متوجه شوید که ترتیب اجرا چقدر مهم است. الگوریتم "rise - and - shine" در ارتباط با مراحلی است که یک شخص از هنگام برخواستن از خواب تا رفتن به سرکار انجام می‌دهد، مراحل: (۱) برخواستن از تختخواب، (۲) پوشیدن لباس راحتی، (۳) دوش گرفتن، (۴) پوشیدن لباس، (۵) خوردن صبحانه، (۶) رفتن به محل کار. این روتین در مورد نحوه انجام کار و ضوابط تصمیم‌گیری می‌تواند موثر باشد حال اگر ترتیب اجرا به صورت زیر جابجا شود:

(۱) برخواستن از تختخواب، (۲) پوشیدن لباس راحتی، (۳) پوشیدن لباس، (۴) دوش گرفتن، (۵) خوردن صبحانه، (۶) رفتن به محل کار. در این حالت شخص مورد نظر، بایستی به محل کار با لباس خیس برود.

مشخص کردن عبارات اجرائی در یک برنامه کامپیوتری، *کنترل برنامه (program control)* نامیده شود، که به اجرای صحیح و مرتب عبارات گفته می‌شود.

۳-٤ شبه‌کد

شبه‌کد (pseudocode) یک زبان مصنوعی و فرمال است که به برنامه‌نویس کمک می‌کند تا الگوریتم خود را توسعه دهد. شبه‌کد مخصوصاً برای ایجاد الگوریتم‌های مفید است که می‌خواهیم آنها را تبدیل به برنامه‌های ساخت یافته کنیم شبه‌کد، همانند زبان روزمره انگلیسی است و مزیت آن درک آسان توسط کاربر است، اگر چه جزء زبان‌های برنامه‌نویسی اجرائی نیست. برنامه‌های شبه‌کد توسط کامپیوتر قابل اجرا نیستند. با این همه، شبه‌کدها، می‌توانند قبل از نوشتن یک برنامه به زبان اجرای مانند C++، برنامه‌نویس را در راه صحیح قرار دهند.

مهندسی نرم‌افزار

شبه‌کد به برنامه‌نویسان کمک می‌کند تا در زمان فرآیند طراحی برنامه یک مفهوم منطقی از برنامه بدست آورند. برنامه شبه‌کد می‌تواند در مرحله بعد به زبان C++ برگردانده شود.

شبه‌کدهایی که از آنها استفاده می‌کنیم، فقط از کاراکترها تشکیل شده‌اند و برنامه‌نویس می‌تواند آنها را در یک برنامه ویرایشگر، تایپ کند. شبه‌کد، فقط شامل عبارات اجرائی است. زمانی که شبه‌کد تبدیل به کدهای C++ شود، برنامه حالت اجرایی پیدا خواهد کرد. اعلان‌ها جزء عبارات اجرائی نیستند. برای مثال، اعلان

```
int i;
```

فقط به کامپایلر نوع متغیر **i** را نشان داده و کامپایلر مکانی در حافظه به این متغیر اختصاص می‌دهد. اما این اعلان‌ها هیچ عمل اجرائی نظیر ورودی، خروجی یا یک عمل محاسباتی را در زمانی که برنامه اجرا می‌شود، از خود نشان نمی‌دهند. تعدادی از برنامه‌نویسان لیستی از متغیرها تهیه کرده و منظور از کاربرد هر متغیر را در ابتدای شبه‌کد قرار می‌دهند.

اکنون به مثالی از شبه کد نگاه می‌کنیم که می‌تواند برای کمک به برنامه‌نویس در ایجاد یک برنامه جمع معرفی شده در شکل ۵-۲ (فصل دوم) نوشته شده باشد. این شبه کد (شکل ۱-۴) متناظر با الگوریتمی است که دو عدد صحیح از کاربر دریافت، آنها را با هم جمع و مجموع آنها را به نمایش در می‌آورد. با اینکه لیست کامل شبه کد را به نمایش درآورده‌ایم، اما شما را با نحوه ایجاد یک شبه کد از صورت مسئله آشنا خواهیم کرد.

```
1  Prompt the user to enter the first integer
2  Input the first integer
3
4  Prompt the user to enter the second integer
5  Input the second integer
6
7  Add first integer and second integer
8  Display result
```

شکل ۱-٤ | شبه کد برنامه جمع شکل ۵-۲.

خطوط 2-1 متناظر با عبارات موجود در خطوط 14-13 از شکل ۵-۲ هستند. دقت کنید که عبارات شبه کد، عبارات ساده انگلیسی هستند که هر وظیفه در C++ را بیان می‌کنند. به همین ترتیب، خطوط 5-4 متناظر با عبارات موجود در خطوط 17-16 و خطوط 8-7 متناظر با عبارات موجود در خطوط 19 و 21 از شکل ۵-۲ هستند.

چندین نکته مهم در شبه کد شکل ۱-۴ وجود دارد. دقت کنید که شبه کد فقط متناظر با کد موجود در تابع **main** است، به این دلیل که معمولاً از شبه کد برای الگوریتم‌ها استفاده می‌شود، نه برای کل برنامه. در این مورد، از شبه کد برای عرضه الگوریتم استفاده شده است. تابع موجود در این کد به اندازه خود الگوریتم مهم نیست. به همین دلیل، خط 23 از شکل ۵-۲ (عبارت **return**) در شبه کد وارد نشده است. عبارت **return** در انتهای هر تابع **main** قرار دارد و در الگوریتم اهمیتی ندارد. سرانجام، خطوط 9-11 از شکل ۵-۲ در شبه کد وجود ندارد چرا که اعلان متغیرها جزء عبارات اجرایی نیستند.

٤-٤ عبارات کنترل

معمولاً، عبارات موجود در یک برنامه یکی پس از دیگری و به ترتیبی که نوشته شده‌اند اجرا می‌شوند، که به اینحالت *اجرای ترتیبی* می‌گویند. انواع متفاوتی از عبارات که بزودی درباره آنها بحث

خواهیم کرد، برنامه‌نویسان را قادر می‌سازند تا با مشخص کردن عبارتی که بایستی قبل از عبارت دیگری اجرا شود، این توالی اجرا را در دست گیرند، که به اینحالت، *کنترل انتقال* می‌گویند.

در دهه ۱۹۶۰، به دلیل وجود مشکلات فراوان در ایجاد نرم‌افزارهای کاربردی، استفاده از روش‌های کنترل انتقال به عنوان یک زمینه بکار گرفته شد و نشانه آن عبارت **goto** بود. این عبارت به برنامه‌نویس اجازه می‌دهد تا کنترل را به یک مکان ویژه در کل برنامه انتقال دهد. عقیده‌ای که برنامه‌نویسی ساخت یافته نام گرفته بود، تقریباً با برنامه‌نویسی حذف **goto** یا کاهش آن (goto-less) مترادف شده است.

با تحقیقات Jacopini و Bohm که نشان داد که برنامه‌ها بایستی بدون **goto** نوشته شوند، دعوت از برنامه‌نویسان به طرف برنامه‌نویسی کاهش استفاده از **goto** آغاز گردید. اما تا سال ۱۹۷۰ برنامه‌نویسی ساخت یافته جدی گرفته نشد. در نتیجه بکار بردن این روش، توسعه نرم‌افزار و کاهش بودجه‌های ساخت آن مشخص شد. کلید تمام این موفقیت‌ها برنامه‌های ساخت‌یافته‌ای بودند که هم وضوح بالاتر و خطاگیری آسانتری داشتند.

Bohm و Jacopini نشان دادند که تمام برنامه‌ها در سه عبارت کنترلی می‌توانند نوشته شوند:

- *عبارت توالی (sequence structure)*
- *عبارت انتخاب (selection structure)*
- *عبارت تکرار (repetition structure)*

ساختار توالی در C++

ساختار توالی بصورت توکار درC++ وجود دارد. مگر اینکه آن را به نحوه دیگری هدایت کنید. کامپیوتر مبادرت به اجرای عبارات C++ یکی پس از دیگری و به ترتیبی که نوشته شده‌اند می‌کند، که این حالت اجرای ترتیبی یا متوالی است. دیاگرام فعالیت (Unified Modeling language) UML به نمایش درآمده در شکل ۲-۴ نشاندهنده یک ساختار توالی است که در آن دو محاسبه به ترتیب انجام می‌شود. زبان C++ اجازه می‌دهد تا به هر اندازه که لازم داریم از ساختار توالی استفاده کنیم. همانطوری که بزودی مشاهده خواهید کرد، در هر کجا که یک عمل می‌تواند موجود باشد، می‌توانیم چندین عمل را به صورت توالی جایگزین کنیم.

در این شکل، دو عبارت مبادرت به افزودن یک نمره به متغیر مجموع (**total**) و مقدار 1 به متغیر **counter** می‌کنند. چنین عباراتی را می‌توان در یک برنامه محاسبه میانگین نمره چند دانشجو مشاهده کرد. برای محاسبه میانگین، مجموع نمرات به تعداد نمرات تقسیم می‌شود. از متغیر شمارنده (**counter**) برای نگهداری تعداد نمرات وارد شده استفاده می‌شود. در بخش ۸-۴ با عبارات مشابهی مواجه خواهید شد.

دیاگرام‌های فعالیت بخشی از UML هستند. یک دیاگرام فعالیت مبادرت به مدل‌سازی *روندکار* (فعالیت) یک بخش از سیستم نرم‌افزاری می‌کند. چنین روندهایی می‌توانند در برگیرنده بخشی از یک الگوریتم، همانند یک ساختار توالی در شکل۲-۴ باشند.دیاگرام‌های فعالیت مرکب از نمادهای معنی‌دار، همانند نمادهای وضعیت عمل (یک مستطیل که گوشه‌های چپ و راست آن به سمت بیرون انحناء داده شده‌اند)، لوزی‌ها و دایره‌های کوچک است. این نمادها توسط *فلش‌های انتقال* به یکدیگر متصل می‌شوند که نشاندهنده روند فعالیت می‌باشند.

همانند شبه کد، دیاگرام‌های فعالیت به برنامه‌نویسان در توسعه و عرضه الگوریتم‌ها کمک می‌کنند، اگر چه برخی از برنامه‌نویسان شبه کد را ترجیح می‌دهند. دیاگرام‌های فعالیت به وضوح نحوه عملکرد ساختارهای کنترل را نشان می‌دهند.

به دیاگرام فعالیت ساختار متوالی در شکل ۲-۴ توجه کنید. این دیاگرام حاوی دو نماد وضعیت عمل است که نشاندهنده اعمالی هستند که اجرا می‌شوند. هر وضعیت عمل حاوی یک بیان‌کننده عمل است، "add grade to total" و "add 1 to counter" که مشخص‌کننده عملی هستند که انجام خواهند شد. سایر اعمال می‌توانند محاسباتی یا ورودی/ خروجی باشند.

شکل ۲-٤ | دیاگرام فعالیت یک ساختار توالی.

فلش‌ها در دیاگرام فعالیت، بنام فلش‌های انتقال شناخته می‌شوند. این فلش‌ها نشاندهنده *انتقال* هستند و بر این نکته دلالت دارند که ترتیب اجرای اعمال به چه صورتی است. برای مثال در شکل ۲-۴ ابتدا **grade** با **total** جمع شده و سپس **1** به **counter** افزوده می‌شود.

دایره توپر ساده که در بالای دیاگرام فعالیت قرار گرفته نشاندهنده وضعیت اولیه فعالیت است، ابتدای روندکار قبل از اینکه برنامه عملیات مدل شده را انجام دهد. دایره توپر احاطه شده با یک دایره توخالی که در پایین دیاگرام فعالیت دیده می‌شود، نشاندهنده وضعیت پایانی است، پایان روندکار پس از اینکه فعالیت‌های خود را انجام داده است.

همچنین شکل ۲-۴ شامل مستطیل‌های با گوشه‌های خم شده به داخل (سمت راست-بالا) است. این مستطیل‌ها، در UML، *نکته* (*note*) نامیده می‌شوند. نکته‌ها توضیحات اضافی در ارتباط با هدف نمادها در دیاگرام ارائه می‌کنند. از نکته‌ها می‌توان در هر دیاگرام UML و نه تنها در دیاگرام‌های فعالیت استفاده کرد. در شکل ۲-۴ از نکته‌های UML برای نمایش کد C++ مرتبط با هر وضعیت عمل در دیاگرام فعالیت استفاده شده است. خط نقطه چین مبادرت به متصل نمودن هر نکته با عنصری می‌کند که در ارتباط با آن توضیح ارائه می‌نماید. معمولاً دیاگرام‌های فعالیت، کد C++ پیاده‌سازی کننده فعالیت را عرضه نمی‌کنند.

اما از این نکته‌ها به این منظور در اینجا استفاده کرده‌ایم تا رابطه دیاگرام با کد C++ مربوطه را بهتر نشان دهیم. برای کسب اطلاعات بیشتر در مورد UML به بخش مبحث آموزشی مهندسی نرم‌افزار که در انتهای فصل ۱ الی ۷، ۹، ۱۰، ۱۲ و ۱۳ قرار دارند مراجعه کرده یا از وب سایت www.uml.org بازدید نمایید.

عبارات انتخاب در C++

زبان C++ سه نوع عبارت انتخاب تدارک دیده است که در مورد آنها در این فصل و فصل بعدی توضیح خواهیم داد. در عبارت انتخاب **if** اگر شرط برقرار باشد عبارت یا عبارات داخل بدنه اجرا شده و در صورتیکه شرط برقرار نباشد از روی آن عبارات بدون اجرای آنها عبور خواهد شد. عبارت **if..else** در صورتیکه شرط برقرار باشد، عبارت (یا دنباله‌ای از عبارات) را انجام داده و در صورتیکه شرط برقرار نباشد، عمل اجرای متفاوتی را به انجام می‌رساند (یا اجرای توالی از عبارات). عبارت **switch** که در فصل ۵ به بررسی آن خواهیم پرداخت با توجه به ارزش یک عبارت، یکی از چندین عمل اجرائی را به اجرا در می‌آورد.

عبارت **if** را *عبارت تک انتخابی* (*single-selection*) می‌نامند، چرا که یک عمل را انتخاب و اجرا یا آنرا رد می‌کند. عبارت **if..else** را *عبارت دو انتخابی* (*double-selection*) می‌نامند، چرا که انتخابی مابین دو حالت متفاوت انجام می‌دهد. عبارت **switch**، *عبارت چند انتخابی* (*multiple-selection*) نامیده می‌شود، چرا که از میان موارد متفاوت انتخاب خود را انجام می‌دهد.

عبارات تکرار در C++

C++ سه نوع عبارت تکرار بنام‌های زیر تدارک دیده است:

- **while**
- **do..while**
- **for**

عبارت تکرار **while**، در این فصل معرفی خواهد شد و عبارات **do..while**، و **for** در فصل ۵ توضیح داده خواهند شد. کلمات **if**، **else**، **switch**، **while**، **do** و **for** همگی جزء کلمات کلیدی C++ هستند (جدول شکل ۳-۴). با بسیاری از این کلمات کلیدی در این کتاب آشنا خواهید شد.

خطای برنامه‌نویسی

استفاده از کلمه کلیدی بعنوان یک شناسه خطای نحوی خواهد بود.

خطای برنامه‌نویسی

نوشتن یک کلمه کلیدی با حروف بزرگ خطای نحوی است تمام کلمات کلیدی در C++فقط شامل حروف کوچک هستند.

C++ Keywords

Keywords common to the C and C++ programming languages

auto	**break**	**case**	**char**	**const**
continue	**default**	**do**	**double**	**else**
enum	**extern**	**float**	**for**	**goto**
if	**int**	**long**	**register**	**return**
short	**signed**	**sizeof**	**static**	**struct**
switch	**typedef**	**union**	**unsigned**	**void**
volatile	**while**			

C++ only keywords

asm	**bool**	**catch**	**class**	**const_cast**
delete	**dynamic_cast**	**explicit**	**false**	**friend**
inline	**mutable**	**namespace**	**new**	**operator**
private	**protected**	**public**	**reinterpret_cast**	**and**
static_cast	**template**	**this**	**throw**	**true**
try	**typeid**	**typename**	**using**	**virtual**
wchar_t	**and_eq**	**bitand**	**bitor**	**export**
not	**not_eq**	**or**	**or_eq**	**xor_eq**

شکل ۳-٤ | کلمات کلیدی C++

خلاصه‌ای بر عبارات کنترلی در C++

C++ دارای سه عبارت کنترلی است، که از این به بعد از آنها بعنوان عبارات کنترلی یاد خواهیم کرد: عبارت توالی، عبارات انتخابی (سه نوع- **if**, **if..else** و **switch**) و عبارات تکرار(سه نوع- **while**, **for** و **do..while**). هر برنامه C++ از ترکیب این عبارات کنترلی ایجاد می‌شود. همانند عبارت توالی در شکل ۲-۴، می‌توانیم هر عبارت کنترلی را بصورت یک دیاگرام فعالیت مدل‌سازی کنیم. هر دیاگرام حاوی یک حالت اولیه و یک حالت پایانی است، که به ترتیب نشاندهنده *نقطه ورودی* (*entry point*) به عبارت کنترلی و *نقطه خروجی* (*exit point*) آن می‌باشند. عبارات کنترلی تک‌ورودی/تک‌خروجی ایجاد آسانتر برنامه‌ها را ممکن می‌سازند. نقطه خروجی یک عبارت کنترلی را می‌توان به نقطه ورودی عبارت کنترلی دیگری

متصل کرد و به همین ترتیب ادامه داد. این فرآیند همانند قرار دادن بلوک‌های بر روی هم است، از اینرو این روش، *عبارت کنترلی پشته* (*control structure stacking*) نام دارد. این روش یکی از روش‌های موجود برای متصل کردن عبارات کنترلی به یکدیگر است. یک روش دیگر *عبارت کنترلی تودرتو* یا *آشیانه‌ای* (*control structure nesting*) می‌باشد که در آن یک عبارت کنترلی می‌تواند در درون عبارت دیگری قرار گیرد. بنابر این الگوریتم‌ها در برنامه‌های C++ فقط متشکل از سه نوع عبارت کنترلی ترکیب شده با این دو روش هستند.

مهندسی نرم‌افزار

هر برنامه را می‌توان با هفت نوع عبارت کنترلی ایجاد کرد (توالی، ***if..else if*** *،* ***switch*** *،* ***while*** *،* ***do..while*** *و* ***for****) که به دو روش با هم ترکیب می‌شوند (عبارت کنترلی پشته‌ای و تو در تو یا آشیانه‌ای).*

٤-٥ عبارت انتخاب if

در یک عبارت انتخاب، هدف برگزیدن یکی از گزینه‌های موجود برای انجام آن است. برای مثال، فرض کنید که شرط قبولی در یک امتحان نمره 60 است از (100). عبارت شبه‌کد آن بصورت زیر می‌باشد:

If student's grade is greater than or equal to 60
Print "Passed"

شرط "student's grade is greater than or equal to 60" می‌تواند برقرار باشد یا نباشد. اگر شرط برقرار باشد عبارت "Passed" به معنی قبول شدن به نمایش در می‌آید و عبارت پس از شبه‌کد به ترتیب اجرا می‌شود (بیاد داشته باشید که شبه‌کد یک زبان برنامه‌نویسی واقعی نیست). اگر شرط برقرار نباشد عبارت چاپ نادیده گرفته می‌شود و عبارت شبه‌کد بعدی به ترتیب اجرا خواهد شد. عبارت موجود در بدنه عبارت **if** رشته **"Passed"** را به چاپ می‌رساند. همچنین به دندانه‌دار بودن این عبارت در این عبارت انتخاب دقت کنید. دندانه گذاری امری اختیاری است، اما بکارگیری آن بسیار توصیه می‌شود چرا که ارتباط عبارتهای مختلف برنامه را بخوبی نشان می‌دهند. کامپایلر C++ کاراکترهای *whitespace* یعنی کاراکترهای فاصله، tab و خطوط جدید بکار رفته در ایجاد دندانه‌ها و فاصله‌گذاری عمودی را بجز کاراکترهای whitespace بکار رفته در رشته‌ها، در نظر نمی‌گیرد.

برنامه‌نویسی ایده‌ال

سعی کنید از روش دندانه‌گذاری ثابتی در برنامه‌های خود استفاده کنید تا خوانایی برنامه افزایش یابد.

می‌توان این عبارت شبه‌کد **if** را در زبان C++ بصورت زیر نوشت

```
if (grade >= 60 )
```

```
cout << "Passed";
```

اگر به کد C++ دقت کنید متوجه شباهت نزدیک آن با شبه‌کد خواهید شد و نقش شبه‌کد به عنوان یک ابزار توسعه برنامه بخوبی آشکار می‌شود.

در شکل ۴-۴ دیاگرام فعالیت عبارت تک‌انتخابی **if** نشان‌داده شده است. این دیاگرام حاوی یکی از مهمترین نمادها در یک دیاگرام فعالیت است. نماد لوزی یا *نماد تصمیم* نشان می‌دهد که باید در آن نقطه تصمیمی اتخاذ گردد. نماد تصمیم‌گیری بر این نکته دلالت دارد که روند کار در امتداد مسیری به کار ادامه خواهد داد که توسط نماد وابسته *نگهبان شرط* تعیین می‌شود (آیا شرط برقرار است یا خیر). هر فلش یا بردار انتقال خارج شده از یک نماد تصمیم دارای یک نگهبان شرط است (در درون براکت‌های مربعی در بالا یا کنار فلش انتقال جای می‌گیرد). اگر شرط یک نگهبان شرط برقرار باشد، روند کار وارد وضعیت عملی می‌شود که فلش انتقال به آن اشاره می‌کند. در شکل ۴-۴ اگر grade بزرگتر یا برابر 60 باشد، برنامه کلمه "Passed" را بر روی صفحه نمایش چاپ کرده و سپس انتقال به وضعیت پایانی در این فعالیت می‌رسد. اگر grade کوچکتر از 60 باشد، بلافاصله برنامه به وضعیت پایانی منتقل می‌شود، بدون اینکه پیغامی چاپ کند.

شکل ٤-٤ | دیاگرام فعالیت عبارت if.

در فصل اول آموختیم، تصمیم‌گیری می‌تواند براساس شرط‌هایی صورت گیرد که حاوی عملگرهای رابطه‌ای یا برابری هستند. در واقع، در C++ یک شرط می‌تواند بر پایه هر عبارتی ارزیابی گردد، اگر عبارت با صفر ارزیابی شود، با آن همانند *false* (عدم برقراری شرط) رفتار خواهد شد و اگر عبارت با مقداری غیر از صفر ارزیابی گردد، با آن همانند *true* (برقراری شرط) رفتار می‌شود. زبان C++ دارای نوع داده *بولی*(*bool*) برای متغیرهایی است که فقط قادر به نگهداری مقادیر **true** و **false** هستند، که هر دو جزء کلمات کلیدی در C++ می‌باشند.

قابلیت حمل

برای حفظ سازگاری با نسخه‌های قبلی C که از اعداد صحیح برای مقادیر بولی استفاده می‌کردند، می‌توان برای عرضه یک مقدار بولی **true** *از هر مقدار غیرصفری استفاده کرد (معمولاً کامپایلرها از* **1** *استفاده می‌کنند). برای عرضه یک مقدار بولی* **false** *نیز می‌توان از مقدار صفر استفاده کرد.*

دقت کنید که عبارت **if** یک عبارت *تک‌ورودی/ تک‌خروجی* است. همچنین دیاگرام‌های مابقی عبارات کنترلی نیز حاوی نمادهای وضعیت اولیه، فلش‌های انتقال، وضعیت اجرا، تصمیم‌گیری و وضعیت پایانی هستند. به این نحوه نمایش عبارات کنترلی روش مدل برنامه‌نویسی *اجرائی/تصمیم‌گیری* گفته می‌شود.

می‌توانیم هفت صندوق را تصور کنیم که هر کدام فقط حاوی دیاگرام‌های فعالیت UML خالی از یکی از هفت نوع عبارت (ساختار) کنترلی است. وظیفه برنامه‌نویس جفت‌وجور کردن برنامه با سرهمبندی کردن دیاگرام فعالیت به هر تعداد از هر نوع عبارت کنترلی تصریح شده در الگوریتم است که فقط به دو روش قابل انجام است، روش *پشته‌ای* و *تودرتو*. در ادامه وضعیت‌های عمل و تصمیم‌گیری را با عبارات اجرایی و نگهبانان شرط به روش مقتضی پر می‌کند. در ارتباط با نوشتن انواع روش‌هایی که می‌تواند در عبارات اجرایی و تصمیم‌گیری بکار گرفته شوند، صحبت خواهیم کرد.

شکل ٤-٤ | دیاگرام فعالیت عبارت تک انتخابی if.

٤-٦ عبارت انتخاب if..else

همانطوری که گفته شد عبارت انتخاب **if** فقط در صورت برقرار بودن شرط، عملی را به اجرا در می‌آورد، در غیر اینصورت از روی عبارت یا عبارات پرش می‌کند. عبارت انتخاب **if..else** این امکان را به برنامه‌نویس می‌دهد که تعیین کند چه اعمالی در برقرار بودن شرط اجرا شوند و چه اعمالی در حالت برقرار نبودن شرط به اجرا در آیند. برای مثال، در شبه‌کد زیر

```
If Student's grade is greater than or equal to 60
        Print "Passed"
Else
        Print "Failed"
```

اگر نمره دانش‌آموز برابر ۶۰ یا بالاتر باشد، عبارت "*Passed*" به نمایش در می‌آید و اگر کمتر از آن باشد عبارت "*Failed*". در هر دو حالت پس از انجام عمل چاپ، عبارت شبه‌کد بعدی به اجرا گذاشته خواهد شد.

عبارت شبه‌کد **if..else** مطرح شده را می‌توان در زبان C++ و به فرم زیر نوشت:

```
if (grade >= 60 )
     cout << "Passed";
else
     cout << "Failed";
```

به دندانه‌دار بودن بدنه شرط **else** دقت کنید که با خطوط بالای خود در شرط **if** یکسان قرار گرفته‌اند.

برنامه‌نویسی ایده‌ال

دندانه‌دار نوشتن هر دو قسمت بدنه عبارت ***if..else*** *خوانائی برنامه را افزایش می‌دهد.*

در شکل ۵-۴ روند کنترل جریان در یک عبارت (ساختار) **if..else** نشان داده شده است. مجدداً توجه کنید (در کنار وضعیت اولیه، فلش‌های انتقال و وضعیت پایانی) که نمادهای بکار رفته در این دیاگرام فعالیت عبارتند از نمادهای عمل و تصمیم‌گیری و تاکید ما بر مدل‌سازی عمل/ تصمیم‌گیری است. مجدداً

به صندوق‌های خالی از دیاگرام‌های فعالیت عبارات انتخاب دوگانه فکر کنید که برنامه‌نویس می‌تواند به روش پشته‌ای یا تودرتو با سایر دیاگرام‌های فعالیت ساختارهای کنترلی بکار گیرد تا مبادرت به پیاده‌سازی الگوریتم کند.

شکل ٥-٤ دیاگرام فعالیت عبارت دو انتخابی if..else

عملگر شرطی (:?)

زبان C++ حاوی عملگر شرطی (**:?**)، است که قرابت نزدیکی با عبارت **if..else** دارد. عملگر شرطی C++ تنها عملگر *ternary* است، به این معنی که سه عملوند دریافت می‌کند. عملوندها به همراه عملگر شرطی تشکیل *عبارت شرطی* را می‌دهند. عملوند اول نشاندهنده شرط می‌باشد، عملوند دوم مقداری است که در صورت **true** بودن شرط انتخاب می‌شود و عملوند سوم مقداری است که در صورت برقرار نبودن شرط یا **false** بودن آن انتخاب می‌شود. برای مثال عبارت زیر

```
cout << (grade >= 60 ? "Passed" : "Failed" );
```

حاوی یک عبارت شرطی، است که در صورت برقرار بودن شرط **grade >= 60** رشته **"Passed"** ارزیابی می‌شود، اما اگر شرط برقرار نباشد، رشته **"Failed"** بکار گرفته خواهد شد. از اینرو عملکرد این عبارت شرطی دقیقا همانند عملکرد عبارت **if..else** قبلی است. عملگر شرطی از تقدم پایین‌تری برخوردار است و از اینرو معمولا کل عبارت شرطی را در درون پرانتزها قرار می‌دهند.

اجتناب از خطا

برای اجتناب از مشکل اولویت و روشن شدن مطلب، سعی کنید عبارات شرطی (که در عبارات بزرگ شرکت می‌کنند) را در درون پرانتزها قرار دهید.

مقادیر موجود در یک عبارت شرطی قادر به اجرا شدن نیز هستند. برای مثال عبارت شرطی زیر مبادرت به چاپ "**Passed**" یا "**Failed**" می‌کند.

```
grade >= 60 ? cout << "Passed" : cout << "Failed";
```

این عبارت به صورت زیر تفسیر می‌شود «اگر **grade** بزرگتر یا مساوی **60** باشد، پس **cout<<"Passed"**، در غیر اینصورت **cout<<"Faild"**." همچنین این عبارت قابل مقایسه با عبارت if..else قبلی است. عبارات شرطی را می‌توان در مکان‌های از برنامه که امکان استفاده از **if..else** وجود ندارد، بکار گرفت.

عبارات تودرتوی if..else

عبارت تودرتوی **if..else** برای تست چندین شرط با قرار دادن عبارتهای **if..else** در درون عبارتهای **if..else** دیگر است. برای مثال، عبارت شبه‌کد زیر، حرف "A" را برای نمره‌های بزرگتر یا برابر 90، "B" را برای نمره‌های در محدودهٔ 89-80، "C" را برای نمره‌های در محدودهٔ 79-70، "D" را برای نمره‌های در محدودهٔ 69-60 و "F" را سایر نمرات به چاپ می‌رساند.

```
If student's grade is greater then or equal to 90
        Print "A"
Else
        If student's grade is greater than or equal to 80
                Print "B"
        Else
                If student's grade is greater than or equal to 70
                        Print "C"
                Else
                        If student's grade is greater than or equal to 60
                                Print "D"
                Else
                                Print "F"
```

عبارت شبه‌کد بالا را می‌توان در زبان C++ و به فرم زیر نوشت:

```
if (studentGrade>=90) // 90 and above gets "A"
    cout << "A";
else
   if (studentGrade>=80) // 80-89 gets "B"
       cout << "B";
   else
      if (strudentGrade>=70) // 70-79 gets "C"
         cout << "C";
      else
         if (studentGrade>=60) // 60-69 gets "D"
            cout << "D";
         else // less than 60 gets "F"
            cout << "F";
```

اگر مقدار **studentGrade** بزرگتر یا مساوی 90 باشد، اولین شرط از پنج شرط برقرار شده و فقط عبارت **cout** قرار گرفته در بدنه اولین شرط به اجرا در می‌آید. پس از اجرای این عبارت از بخش **else** خارجی عبارت **if..else** عبور خواهد شد.

برنامه‌نویسی ایده‌ال

قبل و بعد از هر عبارت کنترل یک خط خالی قرار دهید تا بتوان عبارتهای کنترل را از سایر نقاط برنامه تشخیص داد.

اکثر برنامه‌نویسان C++ ترجیح می‌دهند که عبارت **if..else** را با استفاده از کلمه کلیدی **else if** و بصورت زیر در برنامه‌های خود بنویسند:

```
if (studentGrade>=90) // 90 and above gets "A"
     cout << "A";
else if (studentGrade>=80) // 80-89 gets "B"
     cout << "B";
else if (studentGrade>=70) // 70-79 gets "C"
     cout << "C";
else if (studentGrade>=60) // 60-69 gets "D"
     cout << "D";
else // less than 60 gets "F"
     cout << "F";
```

هر دو حالت معادل یکدیگرند، اما نوع آخر در نزد برنامه‌نویسان از محبوبیت بیشتری برخوردار است. چرا که از دندانه‌دار کردن عمیق کد به طرف راست اجتناب می‌شود.

کارائی

یک عبارت ***if..else*** *تودرتو می‌تواند بسیار سریعتر از یک سری عبارت* ***if*** *عمل کند، چرا که احتمال دارد شرطی در ابتدای عبارت* ***if..else*** *برقرار شود و کنترل برنامه زودتر از این قسمت خارج گردد.*

کارائی

در یک عبارت ***if..else*** *تودرتو، شرط‌هایی که امکان برقرار شدن* ***(true)*** *آنها بیشتر است در ابتدای عبارت* ***if..else*** *تودرتو قرار دهید. در اینحالت امکان اجرای سریعتر عبارت* ***if..else*** *فراهم می‌آید.*

مشکل dangling-else

همیشه کامپایلر C++ یک **else** را با یک **if** در نظر می‌گیرد، مگر اینکه خلاف آنرا با استفاده از براکت‌ها مشخص کنید. به این مشکل *dangling-else* می‌گویند. برای مثال،

```
if ( x > 5 )
   if ( y > 5 )
       cout << "x and y are > 5";
else
   cout << "x is <=5";
```

به نظر می‌رسد بر این نکته دلالت دارد که اگر **x** بزرگتر از 5 باشد، عبارت **if** تودرتو تعیین می‌کند که آیا **y** نیز بزرگتر از 5 است یا خیر. اگر چنین باشد، رشته "x and y are > 5" در خروجی چاپ می‌شود. در غیر اینصورت اگر **x** بزرگتر از 5 نباشد، بخش **else** از عبارت **if**..**else** رشته "x is <=5" را چاپ خواهد کرد.

با این همه امکان دارد عبارت **if** تودرتوی فوق مطابق با انتظار کار نکند. تفسیر کامپایلر از عبارت بصورت زیر خواهد بود

```
if ( x > 5 )
   if ( y > 5 )
      cout << "x and y are > 5";
   else
  cout << "x is <=5";
```

که در آن بدنه اولین عبارت **if** یک عبارت **if..else** تودرتو است. این عبارت مبادرت به تست بزرگتر بودن **x** از 5 می‌کند. اگر چنین باشد، اجرا با تست **y** بزرگتر از 5 ادامه می‌یابد. اگر شرط دوم برقرار باشد، رشته "x and y are >5" به نمایش در خواهد آمد. با این همه اگر شرط دوم برقرار نباشد، رشته "x is <=5" به نمایش در می‌آید، حتی اگر بدانیم که **x** بزرگتر از 5 است.

برای اینکه عبارت فوق بنحوی کار کند که از انتظار داریم، بایستی کل عبارت بصورت زیر نوشته شود:

```
if ( x > 5 )
{
   if ( y > 5 )
      cout << "x and y are > 5";
}
   else
  cout << "x is <=5";
```

براکت‌ها به کامپایلر نشان می‌دهند که دومین **if** در بدنه اولین **if** قرار دارد و **else** در ارتباط با اولین **if** می‌باشد.

بلوک‌ها

معمولا عبارت انتخاب **if** فقط منتظر یک عبارت در بدنه خود است. به همین ترتیب، هر یک از بخش‌های **else** و **if** در یک عبارت **if..else** انتظار مقابله با یک عبارت در بدنه خود را دارند. برای وارد کردن چندین عبارت در بدنه یک **if** یا در بخش‌های **if..else** ، عبارات را در درون براکت‌ها (**{ }**) قرار دهید. به مجموعه‌ای از عبارات موجود در درون یک جفت براکت، *بلوک* می‌گویند.

مهندسی نرم‌افزار

یک بلوک می‌تواند در هر کجای برنامه که یک عبارت منفرد می‌تواند در آنجا قرار داده شود، جای گیرد.

مثال زیر شامل یک بلوک در بخشی از **else** یک عبارت **if..else** است.

```
if (studentGrade>=60)
    cout << "Passed.\n";
else
{
   cout << "Failed.\n";
   cout << "You must take this course again.\n"
}
```

در این مورد، اگر **studentGrade** کمتر از 60 باشد، برنامه هر دو عبارت موجود در بدنه **else** را اجرا کرده و پیغام‌های زیر را چاپ می‌کند.

```
Failed
You must take this course agein.
```

به براکت‌های احاطه‌کننده دو عبارت در ضابطه **else** دقت کنید. این براکت‌ها مهم هستند. بدون این براکت‌ها، عبارت

```
cout << "You must take this course again.\n";
```

در خارج از بدنه بخش **else** قرار می‌گیرد و صرفنظر از اینکه شرط برقرار باشد یا خیر، اجرا خواهد شد. این مثال نمونه‌ای از یک خطای منطقی است.

خطای برنامه‌نویسی

فراموش کردن یک یا هر دو براکت که تعیین محدودهٔ یک بلوک هستند، می‌تواند موجب خطای نحوی یا منطقی در برنامه شود.

برنامه‌نویسی ایده‌ال

*همیشه از براکت‌ها در عبارت **if..else** (یا هر عبارت کنترلی) استفاده کنید تا جلوی حوادث ناخواسته گرفته شود، بویژه به هنگام افزودن عباراتی به یک **if** یا ضابطه **else** در ادامه کار. برای اینکه جلوی فراموش‌کاری گرفته شود، برخی از برنامه‌نویسان در همان ابتدای کار و قبل از اینکه حتی عبارتی تایپ کرده باشد، براکت‌های شروع و پایین را تایپ می‌کنند و سپس عبارات مورد نظر را در درون آنها قرار می‌دهد.*

همانند یک بلوک که می‌تواند در هر کجای که یک عبارت منفرد وجود دارد جایگزین گردد، همچنین امکان داشتن *عبارت null* (یا عبارت تهی) وجود دارد. عبارت تهی با جایگزین کردن یک سیمکولن (;) بجای یک عبارت مشخص می‌شود.

خطای برنامه‌نویسی

*قرار دادن یک سیمکولن بلافاصله پس از شرط در یک عبارت **if** موجب رخ دادن خطای منطقی در عبارات **if** تک انتخابی و خطای نحوی در عبارات **if..else** دو انتخابی خواهد شد.*

۷-٤ عبارت تکرار while

یک عبارت تکرار به برنامه‌نویس امکان می‌دهد تا بر مبنای برقرار بودن یا نبودن مقداری در یک شرط، یک عمل را چندین بار و به تکرار انجام دهد. عبارت شبه‌کد زیر یک فرآیند تکرار شوند را در حین خرید نشان می‌دهد:

While there are more items on my shopping list
Purchase next item and cross it off my list

شرط "there are more items on my shopping list" ممکن است برقرار یا برقرار نباشد. اگر شرط برقرار باشد عمل "Purchase next item" و "Cross it off my list" به ترتیب اجرا خواهند شد. این عمل می‌تواند تا زمانی که شرط برقرار است انجام شود. عبارت موجود در ساختار تکرار **while** تشکیل دهنده

بدنه **while** است. سرانجام، زمانیکه شرط برقرار نباشد، تکرار پایان یافته و اولین دستور قرار گرفته پس از عبارت تکرار به اجرا در می‌آید.

مثالی که در زیر آورده شده از عبارت **while** استفاده کرده و اولین توان 3 بزرگتر از 100 را پیدا می‌کند. در ابتدای کار متغیر **product** با **3** مقدار دهی اولیه شده است:

```
int product = 3;

while (product <= 100)
   product = product * 3;
```

هنگامی که برنامه وارد عبارت **while** می‌شود، مقدار متغیر **product** برابر 3 است. متغیر **product** بصورت مکرر در 3 ضرب می‌شود و مقادیر 9، 27، 81 و 243 بدست می‌آیند. زمانیکه مقدار **product** برابر 243 شود، شرط **product<=100** در عبارت **while** برقرار نخواهد شد. در چنین حالتی تکرار با مقدار 243 برای **product** پایان می‌پذیرد. اجرای برنامه با عبارت بعد از **while** دنبال می‌شود. [نکته: اگر شرط یک عبارت **while** در همان ابتدا برقرار نباشد، عبارات قرار گرفته در بدنه عبارت تکرار اجرا نخواهند شد.]

خطای برنامه‌نویسی

شرط عبارت تکرار را به نوعی تنظیم نمائید که برنامه برای همیشه در داخل حلقه قرار نگیرد، در غیر اینصورت برنامه در حلقه بی‌نهایت "infinite loop" گرفتار می‌شود.

دیاگرام فعالیت UML به نمایش درآمده در شکل ۶-۴ نشاندهنده روند کنترلی است که متناظر با عبارت **while** مطرح شده در قسمت فوق می‌باشد. مجدداً نمادهای موجود در این دیاگرام نشاندهنده یک وضعیت عمل و یک تصمیم‌گیری هستند. همچنین این دیاگرام مبادرت به معرفی *نماد ادغام* UML کرده است، که دو روند یا جریان فعالیت را به یک فعالیت متصل می‌کند. UML نماد ادغام و تصمیم‌گیری را بصورت لوزی نشان می‌دهد. در این دیاگرام، نماد ادغام مبادرت به پیوند انتقال‌ها از وضعیت اولیه و از وضعیت عمل کرده است، از اینرو هر دو جریان وارد بخش تصمیم شده‌اند که تعیین می‌کند آیا حلقه مجدداً باید تکرار شود یا خیر. می‌توان نمادهای تصمیم و ادغام را توسط تعداد فلش‌های انتقال «واردشونده» و «خارج‌شونده» تشخیص داد. نماد تصمیم دارای یک فلش انتقال اشاره‌کننده به لوزی داشته و دارای دو یا چند فلش انتقال اشاره‌کننده به خارج از لوزی است که نشاندهنده انتقالات ممکنه از این نقطه هستند. علاوه بر این، هر فلش انتقال اشاره‌کننده به خارج از نماد تصمیم دارای یک نگهبان شرط در کنار خود است. در طرف دیگر، نماد ادغام قرار دارد که دارای دو یا چند فلش انتقال اشاره‌کننده به لوزی است و فقط یک فلش انتقال از آن خارج می‌شود و نشان می‌هد که چندین روند با یکدیگر برای انجام

فعالیت ادغام شده‌اند. توجه کنید، که برخلاف نماد تصمیم، نماد ادغام دارای یک رونوشت در کد C++ نیست.

دیاگرام شکل ۶-۴ بوضوح عملیات تکرار در عبارت **while** مطرح شده در ابتدای این بخش را نشان می‌دهد. فلش انتقال از وضعیت عمل به حالت ادغام اشاره می‌کند، که انتقال را به تصمیم باز می‌گرداند تا تستی دایر بر اینکه آیا حلقه دوباره باید صورت گیرد یا خیر انجام دهد، این حلقه زمانی شکسته می‌شود که شرط **product>100** برقرار گردد. پس از خاتمه عملیات **while**، کنترل به عبارت بعدی در برنامه انتقال می‌یابد (در این مورد وضعیت پایانی).

می‌توانید دیاگرام‌های فعالیت UML خالی عبارت تکرار **while** را تصور کنید که برنامه‌نویسان می‌توانند هر تعداد از آنها را به روش پشته‌ای یا تودرتو با سایر دیاگرام‌های فعالیت عبارات کنترلی بکار گیرند تا بخشی از الگوریتم را پیاده‌سازی کنند.

شکل ٦-٤ | دیاگرام فعالیت UML عبارت تکرار while.

٤-٨ فرموله کردن الگوریتم: شمارنده-کنترل تکرار

برای اینکه با نحوه توسعه الگوریتم‌ها آشنا شوید، مسئله بدست آوردن میانگین نمرات یک کلاس را با روش‌های مختلف بررسی می‌کنیم. صورت مسئله عبارت است از:

از کلاسی با ده دانش‌آموز آزمونی بعمل آمده است. نمرات این آزمون در اختیار شما قرار دارد (نمرات در محدودهٔ 0 تا 100 هستند). مجموع نمرات دانش‌آموزان و میانگین نمرات این کلاس را بدست آورید.

میانگین کلاس عبارت است از مجموع نمرات تقسیم بر تعداد دانش‌آموزان. الگوریتم بکار رفته بر روی کامپیوتر به منظور حل این مسئله بایستی تک تک نمرات را به عنوان ورودی دریافت کرده، محاسبه میانگین را انجام داده و نتیجه را به نمایش در آورد.

الگوریتم شبه‌کد با شمارنده-کنترل تکرار

اجازه دهید تا از شبه‌کد استفاده کرده و لیستی از فعالیت‌های اجرائی تهیه و ترتیب اجرا را مشخص سازیم. از روش *شمارنده-کنترل تکرار* برای دریافت تک تک نمرات بعنوان ورودی استفاده می‌کنیم. در این تکنیک از متغیری بنام *شمارنده* (*counter*) برای تعیین تعداد دفعات مجموعه‌ای از عبارات که اجرا خواهند شد، استفاده می‌شود. روش شمارنده-کنترل تکرار، روش *تکرار تعریف شده* نیز نامیده می‌شود چرا که تعداد تکرار قبل از اینکه حلقه آغاز شود، مشخص است. در این مثال، اجرای حلقه با رسیدن شمارنده به 10 خاتمه می‌یابد. در این بخش به معرفی الگوریتم شبه‌کد (شکل ۷-۴) و نسخه‌ای از کلاس

GradeBook (شکل‌های ۸-۴ و ۹-۴) می‌پردازیم که الگوریتم را توسط یک تابع عضو C++ پیاده‌سازی می‌کند. در بخش ۹-۴ با توسعه الگوریتم‌ها با استفاده از شبه‌کد آشنا خواهید شد.

به نقش *total* و *counter* در الگوریتم شبه‌کد دقت کنید (شکل ۷-۴). در واقع *total* متغیری است که از آن برای محاسبه مجموع مقادیر استفاده می‌شود و *counter* متغیری است که نقش شمارنده بر عهده دارد، در این برنامه، شمارنده تعداد نمرات وارد شدهٔ توسط کاربر را ثبت می‌کند.

Set total to zero
Set grade counter to one
While grade counter is less than or equal to 10
    Input the next grade
    Add the grade to the total
    Add one to the grade counter
Set the class average to the total divided by 10
Print the class average

شکل۷-٤ | الگوریتم شبه‌کد با استفاده از روش شمارنده-کنترل تکرار برای حل مسئله میانگین کلاس.

```
1  // Fig. 4.8: GradeBook.h
2  // Definition of class GradeBook that determines a class average.
3  // Member functions are defined in GradeBook.cpp
4   include <string> // program uses C++ standard string class
5  using std::string;
6
7  // GradeBook class definition
8  class GradeBook
9  {
10 public:
11    GradeBook( string ); // constructor initializes course name
12    void setCourseName( string ); // function to set the course name
13    string getCourseName(); // function to retrieve the course name
14    void displayMessage(); // display a welcome message
15    void determineClassAverage(); // averages grades entered by the user
16 private:
17    string courseName; // course name for this GradeBook
18 }; // end class GradeBook
```

شکل۸-٤ | مسئله میانگین کلاس با استف‌اده از روش شمارنده-کنترل تکرار:سرآیند فایل GradeBook.

```
1  // Fig. 4.9: GradeBook.cpp
2  // Member-function definitions for class GradeBook that solves the
3  // class average program with counter-controlled repetition.
4   include <iostream>
5  using std::cout;
6  using std::cin;
7  using std::endl;
8
9   include "GradeBook.h" // include definition of class GradeBook
10
11 // constructor initializes courseName with string supplied as argument
12 GradeBook::GradeBook( string name )
13 {
14    setCourseName( name ); // validate and store courseName
15 } // end GradeBook constructor
16
17 // function to set the course name;
18 // ensures that the course name has at most 25 characters
19 void GradeBook::setCourseName( string name )
20 {
21    if ( name.length() <= 25 ) // if name has 25 or fewer characters
```

```
22       courseName = name; // store the course name in the object
23    else // if name is longer than 25 characters
24    { // set courseName to first 25 characters of parameter name
25       courseName = name.substr( 0, 25 ); // select first 25 characters
26      cout << "Name \"" << name << "\" exceeds maximum length (25).\n"
27          << "Limiting courseName to first 25 characters.\n" << endl;
28    } // end if...else
29 } // end function setCourseName
30
31 // function to retrieve the course name
32 string GradeBook::getCourseName()
33 {
34    return courseName;
35 } // end function getCourseName
36
37 // display a welcome message to the GradeBook user
38 void GradeBook::displayMessage()
39 {
40    cout << "Welcome to the grade book for\n" << getCourseName() << "!\n"
41       << endl;
42 } // end function displayMessage
43
44 // determine class average based on 10 grades entered by user
45 void GradeBook::determineClassAverage()
46 {
47    int total; // sum of grades entered by user
48    int gradeCounter; // number of the grade to be entered next
49    int grade; // grade value entered by user
50    int average; // average of grades
51
52    // initialization phase
53    total = 0; // initialize total
54    gradeCounter = 1; // initialize loop counter
55
56    // processing phase
57    while ( gradeCounter <= 10 ) // loop 10 times
58    {
59       cout << "Enter grade: "; // prompt for input
60       cin >> grade; // input next grade
61       total = total + grade; // add grade to total
62       gradeCounter = gradeCounter + 1; // increment counter by 1
63    } // end while
64
65    // termination phase
66    average = total / 10; // integer division yields integer result
67
68    // display total and average of grades
69    cout << "\nTotal of all 10 grades is " << total << endl;
70    cout << "Class average is " << average << endl;
71 } // end function determineClassAverage
```

شکل۹-٤ | مسئله میانگین کلاس با استفاده از روش شمارنده-کنترل تکرار:کد منبع فایل GradeBook.

افزایش قابلیت اعتبارسنجی GradeBook

قبل از اینکه به بحث پیاده‌سازی الگوریتم میانگین کلاس بپردازیم، اجازه دهید به بهبود کارائی انجام گرفته بر روی کلاس **GradeBook** توجه کنیم. در برنامه ۱۶-۳، تابع **setCourseName** مبادرت به اعتبارسنجی نام دوره با تست طول نام دور می‌کرد، که باید کمتر یا برابر 25 کاراکتر باشد (با استفاده از یک عبارت **if**). اگر شرط برقرار بود، نام دوره بکار گرفته می‌شد. سپس این کد با یک عبارت **if** دیگر دنبال می‌شد که مبادرت به تست طول نام دوره می‌کرد که آیا بزرگتر از 25 کاراکتر است یا خیر. دقت کنید که شرط عبارت **if** دوم کاملاً متضاد شرط **if** اول است. اگر شرطی با **true** ارزیابی گردد، بایستی

شرط‌های دیگر با **false** ارزیابی شوند. پیاده‌سازی چنین وضعیتی توسط عبارت **if..else** بهتر خواهد بود، از اینرو کد خود را با جایگزین کردن دو عبارت **if** با یک عبارت **if..else** اصلاح کرده‌ایم (خطوط 28-21 از برنامه شکل ۹-۴).

```
1  // Fig. 4.10: fig04_10.cpp
2  // Create GradeBook object and invoke its determineClassAverage function.
3   include "GradeBook.h" // include definition of class GradeBook
4
5  int main()
6  {
7     // create GradeBook object myGradeBook and
8     // pass course name to constructor
9     GradeBook myGradeBook( "CS101 C++ Programming" );
10
11    myGradeBook.displayMessage(); // display welcome message
12    myGradeBook.determineClassAverage(); // find average of 10 grades
13    return 0; // indicate successful termination
14 } // end main
```

```
Welcome to the grade book for
CS101 C++ Programming
  Enter grade: 67
  Enter grade: 78
  Enter grade: 89
  Enter grade: 67
  Enter grade: 87
  Enter grade: 98
  Enter grade: 93
  Enter grade: 85
  Enter grade: 82
  Enter grade: 100

  Total of all 10 grade is 846
  class average is 84
```

شکل ۱۰-٤| برنامه میانگین کلاس با شمارنده-کنترل تکرار:ایجاد یک شی از کلاس GradeBook (شکل ۸-٤ و ۹-٤) و فراخوانی تابع عضو determineClassAverage.

پیاده‌سازی شمارنده-کنترل تکرار در کلاس GradeBook

کلاس **GradeBook** (شکل ۸-۴ و ۹-۴) حاوی یک سازنده (اعلان شده در خط 11 از شکل ۸-۴ و تعریف شده در خطوط 15-12 از شکل ۹-۴) است که مبادرت به تخصیص مقداری به متغیر نمونه کلاس **courseName** می‌کند (اعلان شده در خط 17 از شکل ۸-۴). در خطوط 29-19، 35-32 و 42-38 از شکل ۹-۴ توابع عضو **setCourseName**، **getCourseName** و **displayMessage** تعریف شده‌اند. در خطوط 71-45 تابع عضو **determineClassAverage** تعریف شده است که پیاده سازی‌کننده الگوریتم میانگین کلاس توضیح داده شده در شبه کد شکل ۷-۴ است.

در خطوط 50-47 متغیرهای محلی **total**، **gradeCounter**، **grade** و **average** از نوع **int** اعلان شده‌اند. در متغیر **grade** ورودی کاربر ذخیره می‌شود. توجه کنید که اعلان‌های فوق در بدنه تابع عضو **determineClassAverage** قرار دارند.

در نسخه‌های کلاس **GradeBook** مطرح شده در این فصل، فرآیند خواندن و پردازش نمرات به روش ساده‌ای در نظر گرفته شده‌اند. محاسبه میانگین در تابع عضو **determineClassAverage** و با استفاده از متغیرهای محلی صورت می‌گیرد. در این بخش مبادرت به ذخیره‌سازی نمرات دانشجویان نمی‌کنیم. در فصل هفتم، با تغییری که در کلاس **GradeBook** انجام می‌دهیم قادر به نگهداری نمرات در حافظه خواهیم بود که توسط ساختمان داده *آرایه* صورت می‌گیرد. در اینحالت به یک شی **GradeBook** اجازه داده می‌شود تا محاسبات مختلف را بر روی همان مجموعه از نمرات انجام دهد بدون اینکه کاربر مجبور به وارد کردن همان نمرات به دفعات باشد.

برنامه‌نویسی ایده‌ال

همیشه یک خط خالی مابین بخش اعلان‌ها و عبارات اجرایی قرار دهید. در این صورت بخش اعلان بخوبی در برنامه مشخص شده و خوانایی برنامه افزایش می‌یابد.

در خطوط 54-53 متغیر **total** با **0** و **gradeCounter** با **1** مقداردهی اولیه شده‌اند.دقت کنید که متغیرهای **totoal** و **gradeCounter** قبل از اینکه در محاسبات بکار گرفته شوند، مقداردهی اولیه شده‌اند. معمولاً متغیرهای شمارنده را با یک یا صفر و بر اساس نیاز مقداردهی اولیه می‌کنند. یک متغیر مقداردهی نشده حاوی یک مقدار اشغال (یا مقدار تعریف نشده) است، آخرین مقداری که در مکان حافظه رزرو شده برای متغیر از قبل وجود داشته است. در این برنامه متغیرهای **grade** و **average** که مقدار خود را از طرف ورودی کاربر و محاسبه میانگین بدست می‌آورند، نیازی به مقداردهی اولیه ندارند.

خط 57 مشخص می‌کند که عبارت **while** تا زمانیکه مقدار **gradeCounter** کمتر یا معادل **10** باشد، تکرار خواهد شد. تا زمانیکه شرط برقرار باشد، ساختار **while** عبارات قرار گرفته مابین براکت‌های بدنه خود را تکرار خواهد کرد.

عبارت بکار رفته در خط 59، جمله **"Enter grade:"** را بنمایش در می‌آورد. این خط معادل عبارت شبه‌کد "*Prompt the user to enter the next grade.*" هستند. خط 60 مقدار وارد شده توسط کاربر را خوانده و آنرا در متغیر **grade** ذخیره می‌کند. این خط معادل شبه‌کد *"Input the next grade."* است. بخاطر دارید که متغیر **grade** در ابتدای برنامه مقداردهی اولیه نشده است، به این دلیل که برنامه مقدار **grade** را از کاربر و در هر بار تکرار حلقه اخذ می‌کند. سپس، برنامه مقدار **total** را با مقدار جدید **grade** که

توسط کاربر وارد شده به روز می‌کند (خط **61**). مقدار **grade** با مقدار قبلی **total** جمع شده و نتیجه به **total** تخصیص می‌یابد.

در خط 62 متغیر **gradeCounter** یک واحد افزایش می‌یابد تا نشان دهد یک نمره مورد پردازش قرار گرفته است. اینکار تا زمانیکه شرط موجود در عبارت **while** برقرار نشود، ادامه می‌یابد. پس از اتمام حلقه، در خط 66 نتیجه محاسبه میانگین به متغیر **average** تخصیص می‌یابد. خط **69** پیغام **"Total of all 10 grades is"** و بدنبال آن مقدار متغیر **total** را بنمایش در می‌آورد. سپس در خط 70 پیغامی حاوی رشته **"Class average is"** که بدنبال آن مقدار متغیر **average** آورده شده، به نمایش در می‌آید. تابع عضو **determineClassAverage**، کنترل را به تابع فراخوان برگشت می‌دهد (تابع **main** در شکل ۱۰-۴).

توصیف کلاس GradeBook

شکل ۱۰-۴ حاوی تابع **main** این برنامه است، که یک شی از کلاس **GradeBook** ایجاد و به توصیف قابلیت‌های آن می‌پردازد. در خط 9 از شکل۱۰-۴ یک شی جدید از **GradeBook** بنام **myGradeBook** ایجاد می‌شود. رشته موجود در خط 9 به سازنده **GradeBook** ارسال می‌شود (خطوط ۱۵-۱۲ از شکل ۹-۴). خط 11 از شکل۱۰-۴ تابع عضو **displayMessage** را برای نمایش پیغام خوش‌آمدگویی به کاربر فراخوانی می‌کند. سپس خط 12 تابع عضو **determineClassAverage** را فراخوانی می‌کند تا کاربر بتواند 10 نمره را وارد کرده و سپس میانگین را محاسبه و چاپ می‌کند. تابع عضو، الگوریتم نشان داده شده در شبه کد شکل ۷-۴ را انجام می‌دهد.

نکاتی در ارتباط با تقسیم صحیح و قطع کردن

محاسبه میانگین توسط تابع عضو **determineClassAverage** صورت می‌گیرد که در واکنش به فراخوانی تابع در خط 12 از شکل۱۰-۴ فعال شده و یک عدد صحیح تولید می‌کند. خروجی برنامه نشان می‌دهد که مجموع نمرات در اجرای نمونه برنامه 846 است که به هنگام تقسیم بر 10، باید 84.6 بدست آید، عددی با نقطه اعشار. با این همه، در نتیجه محاسبه **total/10** عدد 84 بدست آمده است (خط 66 از شکل ۹-۴)، چرا که **total** و 10 هر دو مقادیر عددی صحیح هستند. نتیجه تقسیم دو عدد صحیح یک عدد صحیح است که در آن بخش اعشاری بدست آمده از تقسیم حذف می‌گردد (قطع می‌شود). در بخش بعد با نحوه بدست آوردن نتایج اعشاری از محاسبات آشنا خواهید شد.

خطای برنامه‌نویسی

فرض اینکه تقسیم صحیح مبادرت به گرد کردن (بجای قطع کردن) می‌کند می‌تواند نتایج اشتباهی

بدنبال داشته باشد. برای مثال، 4÷7 حاصل 1.75 را در ریاضی بدست می‌دهد، در حالیکه در یک تقسیم صحیح 1 کوتاه شده و 2 در حالت گرد شده تولید می‌کند.

در برنامه شکل ۹-۴، اگر در خط 66 از **gradeCounter** بجای 10 در محاسبه استفاده شود، خروجی این برنامه مقدار اشتباه 76 را نشان خواهد داد. دلیل اینکار در آخرین تکرار عبارت **while** نهفته است که **gradeCounter** به مقدار 11 در خط 62 افزایش یافته است.

خطای برنامه‌نویسی

استفاده از متغیر شمارنده حلقه، در یک عبارت محاسباتی پس از حلقه، معمولاً سبب تولید خطای منطقی بنام off-by-one-error می‌شود.

۹-٤ فرموله کردن الگوریتم‌ها: مراقبت-کنترل تکرار

اجازه دهید تا به مسئله میانگین کلاس بازگردیم و آنرا مجدداً و اینبار بصورت زیر و کلی‌تر تعریف کنیم:

"برنامه محاسبه میانگین کلاس را به نحوی توسعه دهید تا در هر بار اجرای برنامه، به تعداد اختیاری نمره دریافت کرده و محاسبه میانگین بر روی آنها اعمال شود."

در برنامه قبلی، تعداد نمرات از همان ابتدا مشخص بود (10 نمره). در این برنامه، تعداد نمراتی که بعنوان ورودی وارد خواهند شد مشخص نیستند. برنامه باید بر روی تعداد نمرات وارد شده کار کند. چگونه برنامه تشخیص می‌دهد که به گرفتن نمره پایان دهد؟ محاسبات به چه صورتی باید انجام گرفته و میانگین کلاس به نمایش درآید؟

یک راه‌حل برای رفع این مشکل، استفاده از یک مقدار ویژه بنام *مقدار مراقبتی* (*sentinel value*) است که پایان ورود داده‌ها را مشخص می‌کند (همچنین به این مقدار، *مقدار سیگنال*، *مقدار ساختگی* یا *پرچم* نیز می‌گویند). در این روش کاربر اقدام به وارد کردن نمره‌ها کرده و در پایان مقدار مراقبتی تعیین شده را به عنوان اینکه داده‌های ورودی به اتمام رسیده‌اند، وارد می‌سازد. روش مراقبت-کنترل تکرار، روش *تکرار-تعریف‌نشده* نیز نامیده می‌شود چرا که تعداد دفعات تکرار قبل از اجرای حلقه مشخص نیست.

واضح است که مقدار مراقبتی باید به نحوی انتخاب شود که به عنوان یک ورودی معتبر مورد قبول واقع نشود. بدلیل اینکه نمرات امتحان معمولاً منفی نیستند، می‌توانیم از مقدار 1- به عنوان مقدار مراقبتی در این برنامه استفاده کنیم. بنابراین به هنگام اجرای برنامه، نمرات کلاس، می‌توانند ترتیبی مانند 1- و 84، 74، 75، 96، 93 داشته باشند. برنامه باید نمره میانگین کلاس را با استفاده از مقادیر 93، 96، 75، 74 و 84 محاسبه کرده و به نمایش درآورد (1- یک مقدار مراقبتی است و نباید در محاسبه میانگین وارد شود).

خطای برنامه‌نویسی

انتخاب یک مقدار مراقبتی به عنوان یک داده معتبر، موجب رخ دادن خطای منطقی می‌شود.

الگوریتم شبه‌کد به روش مراقبت کنترل تکرار به روش از بالا به پایین، اصلاح گام‌به‌گام: اولین اصلاح

به هنگام بررسی مسائل پیچیده‌ای همانند این برنامه، عرضه الگوریتم شبه‌کد به آسانی امکان پذیر نمی‌باشد. از اینرو به برنامه میانگین کلاس با استفاده از تکنیکی بنام، از بالا به پایین، اصلاح گام‌به‌گام نزدیک می‌شویم که برای ساخت و توسعه برنامه‌های ساخت‌یافته مناسب و ضروری است. شبه‌کدی که در بالاترین سطح (*top*) ارائه می‌شود، عبارت است از:

Determine the class average for the quiz.

این عبارت تابع و هدف اصلی برنامه است که در واقع کاری که باید برنامه انجام دهد را در بردارد. عبارت top جزئیات ناکافی در مورد اینکه برنامه چگونه بایستی نوشته شود در خود دارد. بنابر این به طرف جزئیات برنامه و اصلاح گام به گام پیش می‌رویم. ابتدا عبارت top به قسمت‌های کوچکی تقسیم می‌شود که هر یک به ترتیب وظایفی در برنامه ایفا می‌کنند. نتیجه این تقسیمات در اولین گام می‌تواند چنین باشد:

Initialize Variables
Input, sum and count the quiz grades
Calculate and print the total of all student grades and the class average

در اینجا، با توجه به اینکه فقط از عبارت توالی استفاده شده، لیست انجام مراحل فقط شامل عبارت‌های اجرائی است که به ترتیب یکی پس از دیگری اجرا می‌شوند.

اصلاح گام‌به‌گام مرحله دوم

مرحله بعدی تجزیه برنامه به جزئیات بیشتر (مرحله دوم)، در ارتباط با متغیرها می‌باشد به یک متغیر بنام **total** نیاز است که مجموع اعداد را در خود نگهداری کند و به یک متغیر دیگر بنام **count** که نشان دهد، چه تعدادی از این اعداد مورد پردازش قرار گرفته‌اند. یک متغیر برای دریافت هر نمره از طریق ورودی و یک متغیر برای نگهداری میانگین محاسبه شده مورد نیاز است. عبارت شبه‌کد:

Initialize variable

را می‌توان به عبارات جزئی‌تر زیر تقسیم کرد:

Initialize total to zero
Initialize counter to zero

توجه کنید که فقط متغیرهای **total** و **counter** نیاز به مقداردهی اولیه قبل از بکارگیری دارند. متغیرهای **average** و **grade** (این متغیرها برای محاسبه میانگین و ورودی کاربر استفاده شده است)، نیازی به مقداردهی اولیه ندارند. عبارت شبه‌کد:

Input, sum and count the quiz grades

نیازمند یک عبارت تکرار (حلقه) است که نمرات را دریافت کند. چون بطور دقیق نمی‌دانیم که چه تعداد نمره به عنوان ورودی دریافت خواهیم کرد، از روش مراقبت-کنترل تکرار استفاده می‌کنیم. کاربر در هر زمان یک مقدار معتبر وارد می‌کند و پس از اینکه آخرین مقدار مورد نظر را وارد کرد، مقدار مراقبتی را وارد می‌کند تا از حلقه ورود نمرات خارج شود. برنامه در هر بار که داده وارد می‌شود مقدار آنرا با مقدار مراقبتی مقایسه می‌کند. دومین اصلاح بر روی عبارت شبه‌کد قبلی می‌تواند بصورت زیر باشد:

```
Prompt the user to enter the first grade
Input the first grade (possibly the sentinel)
While the user has not yet entered the sentinel
    Add this grade to the running total
    Add one to the grade counter
    Input the next grade (possibly the sentinel)
```

عبارت شبه‌کد زیر

```
Calculate and print the total of all student grades and the class average
```

می‌تواند به صورت عبارات جزئی‌تر زیر نوشته شود:

```
If the counter is not equal to zero
    Set the average to the total divided by the counter
    Print the total of all student grades in the class
    Print the average
Else
    Print "No grades were entered"
```

توجه کنید که در این قسمت برای جلوگیری از بروز خطای منطقی تقسیم بر صفر، یک تست بکار برده شده است که اگر در برنامه تشخیص داده نشود، می‌تواند مشکل ساز شود. شبه‌کد کامل برنامه میانگین در شکل ۱۱-۴ آورده شده است.

خطای برنامه‌نویسی

نتیجه تقسیم بر صفر خطای عظیم در زمان اجرا است.

اجتناب از خطا

به هنگام اجرای یک عمل تقسیم بر عبارتی که ممکن است مقدار آن صفر باشد، باید تستی به همین منظور و رسیدگی به آن در برنامه تدارک دیده شود. رسیدگی به این امر می‌تواند چاپ یک پیغام ساده خطا باشد. گاهی اوقات انجام عملیات پیچیده مورد نیاز است.

```
Initialize total to zero
Initialize counter to zero
Input the first grade (possibly the sentinel)
While the user has not as yet entered the sentinel
    Add this grade to the running total
    Add one to the grade counter
    Input the next grade (possibly the sentinel)
```

```
If the counter is not equal to zero
    Set the average to the total divided by the counter
    Print the average
Else
    Print "No grades were entered"
```

شکل ۱۱-٤ | الگوریتم شبه‌کد با استفاده از روش مراقبت-کنترل تکرار برای حل مسئله میانگین کلاس.

برنامه‌نویسی ایده‌ال

با قراردادن خطوط خالی در برنامه‌های شبه‌کد خوانائی آنها افزایش می‌یابد. خطوط خالی موجب می‌شوند تا عبارتهای کنترلی شبه‌کد و فازهای برنامه از هم متمایز شوند.

مهندسی نرم‌افزار

بسیاری از الگوریتم‌ها را می‌توان به صورت منطقی به سه فاز تقسیم کرد: فاز مقدار دهی که در آن متغیرهای برنامه مقداردهی اولیه می‌شوند، فاز پردازش که مقادیر داده‌ها وارد شده و متغیرها براساس آنها تنظیم می‌شوند و فاز پایان که مرحله انجام محاسبات و چاپ نتایج است.

الگوریتم شبه‌کد ۱۱-۴ مسئله میانگین کلاس را که در ابتدای این بخش بصورت کلی بیان شده بود، برطرف می‌کند. این الگوریتم فقط پس از طی دو مرحله اصلاح گام به گام توسعه یافت، در حالیکه گاهی اوقات به انجام مراحل بیشتر نیاز است.

مهندسی نرم‌افزار

برنامه‌نویسان زمانی به فرآیند از بالا به پایین و اصلاح گام به گام پایان می‌دهند که الگوریتم شبه‌کد بصورت مشخص جزئیات را بیان کرده باشد، به نحوی که بتوان آنها را به برنامه C++ تبدیل کرد. در اینحالت پیاده سازی برنامه C++ براحتی می‌تواند صورت گیرد.

پیاده‌سازی کلاس GradeBook به روش مراقبت-کنترل تکرار

شکل‌های ۱۲-۴ و ۱۳-۴ کلاس **GradeBook** را به نحوی نشان می‌دهند که حاوی تابع عضو **determineClassAverage** است که الگوریتم شبه کد شکل ۱۱-۴ را پیاده‌سازی می‌کند (این کلاس در شکل ۱۴-۴ توصیف شده است). اگر چه هر نمره وارد شده یک عدد صحیح است، امکان تولید یک عدد اعشاری به هنگام محاسبه میانگین وجود دارد، به عبارتی یک عدد حقیقی یا عدد با نقطه اعشار (همانند 7.33، 0.0975 یا 1000.12345). نوع داده **int** نمی‌تواند چنین اعدادی را عرضه کند، از اینرو این کلاس باید از نوع داده دیگری استفاده کند. زبان C++ دارای چندین نوع داده برای ذخیره‌سازی اعداد اعشاری در حافظه است، نوع‌های همانند **float** و **double**. تفاوت اصلی مابین این نوع در این است که در مقایسه با متغیرهای **float**، متغیرهای **double** قادر به نگهداری اعداد بزرگتر و دقیق‌تر در سمت نقطه اعشار هستند، در نتیجه دقت عدد بیشتر خواهد بود. این برنامه مبادرت به معرفی یک عملگر ویژه بنام عملگر *cast* است

که محاسبه میانگین را مجبور می‌کند تا نتیجه را بصورت عدد اعشاری تولید کند. این ویژگی به هنگام بررسی برنامه توضیح داده خواهد شد.

در این مثال مشاهده می‌کنید که عبارات کنترلی می‌توانند به صورت پشته یکی بر روی دیگری قرار داده شوند (بصورت متوالی). عبارت **while** در خطوط 75-67 از شکل۱۳-۴ بلافاصله پس از عبارات **if..else** قرار گرفته است و حالت توالی دارد. قسمت اعظم کد بکار رفته در این مثال با کد برنامه ۹-۴ یکسان است، از اینرو تمرکز خود را بر روی ویژگی‌ها و مباحث جدید متمرکز می‌کنیم.

```
1  // Fig. 4.12: GradeBook.h
2  // Definition of class GradeBook that determines a class average.
3  // Member functions are defined in GradeBook.cpp
4  #include <string> // program uses C++ standard string class
5  using std::string;
6
7  // GradeBook class definition
8  class GradeBook
9  {
10 public:
11    GradeBook( string ); // constructor initializes course name
12    void setCourseName( string ); // function to set the course name
13    string getCourseName(); // function to retrieve the course name
14    void displayMessage(); // display a welcome message
15    void determineClassAverage(); // averages grades entered by the user
16 private:
17    string courseName; // course name for this GradeBook
18 }; // end class GradeBook
```

شکل ۱۲-٤ | برنامه میانگین کلاس با روش مراقبت-کنترل تکرار: فایل سرآیند GradeBook

```
1  // Fig. 4.13: GradeBook.cpp
2  // Member-function definitions for class GradeBook that solves the
3  // class average program with sentinel-controlled repetition.
4  #include <iostream>
5  using std::cout;
6  using std::cin;
7  using std::endl;
8  using std::fixed; // ensures that decimal point is displayed
9
10 #include <iomanip> // parameterized stream manipulators
11 using std::setprecision; // sets numeric output precision
12
13 // include definition of class GradeBook from GradeBook.h
14 #include "GradeBook.h"
15
16 // constructor initializes courseName with string supplied as argument
17 GradeBook::GradeBook( string name )
18 {
19    setCourseName( name ); // validate and store courseName
20 } // end GradeBook constructor
21
22 // function to set the course name;
23 // ensures that the course name has at most 25 characters
24 void GradeBook::setCourseName( string name )
25 {
26    if ( name.length() <= 25 ) // if name has 25 or fewer characters
```

```
27       courseName = name; // store the course name in the object
28    else // if name is longer than 25 characters
29    { // set courseName to first 25 characters of parameter name
30       courseName = name.substr( 0, 25 ); // select first 25 characters
31       cout << "Name \"" << name << "\" exceeds maximum length (25).\n"
32          << "Limiting courseName to first 25 characters.\n" << endl;
33    } // end if...else
34 } // end function setCourseName
35
36 // function to retrieve the course name
37 string GradeBook::getCourseName()
38 {
39    return courseName;
40 } // end function getCourseName
41
42 // display a welcome message to the GradeBook user
43 void GradeBook::displayMessage()
44 {
45    cout << "Welcome to the grade book for\n" << getCourseName() << "!\n"
46       << endl;
47 } // end function displayMessage
48
49 // determine class average based on 10 grades entered by user
50 void GradeBook::determineClassAverage()
51 {
52    int total; // sum of grades entered by user
53    int gradeCounter; // number of grades entered
54    int grade; // grade value
55    double average; // number with decimal point for average
56
57    // initialization phase
58    total = 0; // initialize total
59    gradeCounter = 0; // initialize loop counter
60
61    // processing phase
62    // prompt for input and read grade from user
63    cout << "Enter grade or -1 to quit: ";
64    cin >> grade; // input grade or sentinel value
65
66    // loop until sentinel value read from user
67    while ( grade != -1 ) // while grade is not -1
68    {
69       total = total + grade; // add grade to total
70       gradeCounter = gradeCounter + 1; // increment counter
71
72       // prompt for input and read next grade from user
73       cout << "Enter grade or -1 to quit: ";
74       cin >> grade; // input grade or sentinel value
75    } // end while
76
77    // termination phase
78   if ( gradeCounter != 0 ) // if user entered at least one grade...
79    {
80       // calculate average of all grades entered
81       average = static_cast< double >( total ) / gradeCounter;
82
83       // display total and average (with two digits of precision)
84       cout << "\nTotal of all " << gradeCounter << " grades entered is "
85          << total << endl;
```

```
86       cout<< "Class average is" << setprecision( 2 ) << fixed << average
87          << endl;
88    } // end if
89    else // no grades were entered, so output appropriate message
90       cout << "No grades were entered" << endl;
91 } // end function determineClassAverage
```

شکل ۱۳-٤ | برنامه میانگین کلاس با روش مراقبت-کنترل تکرار: فایل کد منبع GradeBook

```
1  // Fig. 4.14: fig04_14.cpp
2  // Create GradeBook object and invoke its determineClassAverage function.
3
4  // include definition of class GradeBook from GradeBook.h
5  #include "GradeBook.h"
6
7  int main()
8  {
9     // create GradeBook object myGradeBook and
10    // pass course name to constructor
11    GradeBook myGradeBook( "CS101 C++ Programming" );
12
13    myGradeBook.displayMessage(); // display welcome message
14    myGradeBook.determineClassAverage(); // find average of 10 grades
15    return 0; // indicate successful termination
16 } // end main
```

```
Welcome to the grade book for
CS101 C++ Programming

Enter grade or -1 to quit: 97
Enter grade or -1 to quit: 88
Enter grade or -1 to quit: 72
Enter grade or -1 to quit: -1

Total of all 3 grades entered is 257
Class average is 85.67
```

شکل۱٤-٤|برنامه میانگین کلاس با روش مراقبت-کنترل تکرار: ایجاد یک شی از کلاس GradeBook(شکل۱۲-٤ و ۱۳-٤) و فراخوانی تابع عضو determineClassAverage.

در خط 55 متغیر **average** از نوع **double** اعلان شده است. این نوع به محاسبه میانگین امکان می‌دهد تا بصورت یک عدد اعشاری در متغیر ذخیره گردد. در خط 59 متغیر **gradeCounter** با صفر مقداردهی شده چرا که هنوز نمره‌ای وارد نشده است، به یاد داشته باشید که این برنامه از روش مراقبت-کنترل تکرار استفاده می‌کند. به منظور ثبت دقیق تعداد نمرات وارد شدهٔ، متغیر **gradeCounter** فقط به هنگام وارد شدن یک نمره معتبر بعنوان ورودی، افزایش می‌یابد.

تفاوت‌های موجود مابین روش‌های مراقبت-کنترل تکرار و شمارنده-کنترل تکرار

به تفاوت‌های موجود میان روش مراقبت-کنترل تکرار در این برنامه و شمارنده-کنترل تکرار در برنامه ۹-۴ توجه کنید. در روش شمارنده-کنترل تکرار، در هر بار تکرار عبارت **while** (خطوط 63-57 از شکل ۹-۴) یک مقدار از سوی کاربر دریافت می‌گردید. در روش مراقبت-کنترل تکرار، قبل از اینکه برنامه به عبارت **while** برسد، یک مقدار (خطوط 64-63 از شکل ۱۳-۴)) دریافت می‌شود. این مقدار تعیین می‌کند که آیا جریان کنترل برنامه وارد بدنه عبارت **while** شود یا خیر. اگر شرط عبارت **while** برقرار

نباشد (کاربر مقدار مراقبتی وارد کرده باشد)، بدنه عبارت **while** اجرا نخواهد شد (هیچ نمره‌ای وارد نمی‌شود). از سوی دیگر، اگر شرط برقرار شود، بدنه اجرا شده و مقدار وارد شدهٔ کاربر بکار گرفته می‌شود (به **total** افزوده می‌شود، خط 69). پس از پردازش مقدار، مقدار بعدی قبل از اینکه برنامه به انتهای بدنه عبارت **while** برسد توسط کاربر وارد می‌شود (خطوط 74-73). زمانیکه برنامه به **{** در خط 75 می‌رسد، اجرا با تست بعدی در شرط عبارت **while** ادامه می‌یابد (خط 67). مقدار جدید وارد شده تعیین می‌کند که آیا عبارت بدنه **while** مجدداً اجرا شود یا خیر. دقت کنید که مقدار بعدی همیشه قبل از اینکه شرط عبارت **while** ارزیابی شود، بلافاصله توسط کاربر وارد می‌شود. در اینحالت برنامه می‌تواند قبل از اینکه اقدام به پردازش مقداری نماید، تعیین کند که آیا آن مقدار، مقدار مراقبتی است یا خیر. اگر مقدار مراقبتی باشد، عبارت **while** خاتمه می‌یابد و مقدار به **total** افزوده نمی‌شود.

پس از خاتمه حلقه، عبارت if..else در خطوط 90-78 اجرا می‌شود. شرط موجود در خط 78 تعیین می‌کند که آیا نمره‌ای وارده شده است یا خیر. اگر نمره‌ای وارد نشده باشد، بخش else (خطوط 90-89) از عبارت if..else اجرا شده و پیغام "No grades were entered" را به نمایش درآورده و تابع عضو کنترل را به تابع فراخوان برگشت می‌دهد.

به بلوک موجود در حلقه while شکل ۱۳-۴ دقت کنید. بدون حضور براکت‌ها، سه عبارت آخر در بدنه حلقه در خارج از حلقه جای می‌گرفتند و این سبب می‌شد که کامپیوتر این کد را بصورت زیر و نادرست تفسیر کند:

```
// loop until sentinel value read from user
   while ( grade != -1 )
      total = total + grade; // add grade to total
   gradeCounter = gradeCounter + 1; // increment counter

   // prompt for input and read next grade from user
      cout << "Enter grade or -1 to quit: ";
      cin >> grade;
```

در این حالت برنامه دچار یک حلقه بی‌نهایت می‌شود در صورتیکه کاربر 1- را به عنوان اولین نمره وارد نکند (خط 64).

خطای برنامه‌نویسی

فراموش کردن براکت‌های تعیین‌کننده مرز یک بلوک می‌تواند، سبب‌ساز خطاهای منطقی همانند حلقه‌های بی‌نهایت شود.

برنامه‌نویسی ایده‌ال

در حلقه کنترل مقدار مراقبتی، که مقداری از کاربر تقاضا می‌کند، باید مقدار مراقبتی به کاربر نشان داده شود.

دقت اعداد اعشاری و نیاز حافظه

متغیرهای از نوع **float** عرضه‌کننده اعداد با دقت منفرد در نقطه اعشار هستند و دارای هفت رقم معنی‌دار در سیستم‌های 32 بیتی می‌باشند. متغیرهای از نوع **double** عرضه‌کننده دقت مضاعف در نقطه اعشار هستند. این دقت مستلزم دو برابر حافظه مورد نیاز برای یک متغیر **float** است و دارای 15 رقم معنی‌دار در سیستم‌های 32 بیتی است (تقریباً دو برابر دقیق‌تر از متغیرهای**float**). برای اکثر محاسبات صورت گرفته در برنامه‌ها نوع **float** می‌تواند کافی باشد، اما می‌توانید با استفاده از **double** دقت را تضمین کنید. در برخی از برنامه‌ها، حتی متغیرهای از نوع **double** هم کافی نیستند، برنامه‌هایی که خارج از قلمرو بحث این کتاب هستند. اکثر برنامه‌نویسان برای عرضه اعداد اعشاری از نوع **double** استفاده می‌کنند. در واقع C++ بطور پیش‌فرض با تمام اعداد اعشاری که در کد منبع برنامه تایپ می‌کنید (همانند 7.33 و 0.0975) همانند مقادیر **double** رفتار می‌کند. چنین مقادیری در کد منبع بعنوان ثابت‌های اعشاری شناخته می‌شوند.

غالبا اعداد اعشاری در انجام عملیات تقسیم گسترش زیادی پیدا می‌کنند. برای مثال با تقسیم 10 بر 3، نتیجه ...3.333333 با دنباله‌ای از 3های نامتناهی خواهد بود. کامپیوتر فضای ثابتی برای نگهداری چنین مقادیری در اختیار دارد، از اینرو ذخیره سازی مقادیر اعشاری فقط بصورت تخمینی صورت می‌گیرد.

علیرغم اینکه اعداد اعشاری همیشه 100 درصد دقیق نیستند، اما کاربردهای بسیاری دارند. برای مثال، هنگامی که در مورد حرارت عادی بدن یعنی 98.6 صحبت می‌کنیم، نیازی نیست تا دقت اعشاری آنرا بسیار دقیق بیان کنیم. زمانیکه به درجه حرارت در یک دماسنج نگاه می‌کنیم و آنرا 98.6 می‌خوانیم، ممکن است مقدار دقیق آن 98.5999473210643 باشد. اما استفاده از مقدار 98.6 به صورت تخمینی در بسیاری از موارد می‌تواند مناسب و کاربردی باشد.

خطای برنامه‌نویسی

استفاده از اعداد اعشاری با فرض اینکه این اعداد نشاندهنده مقدار کاملاً دقیق هستند (بویژه در عبارات مقایسه‌ای) می‌تواند نتایج اشتباهی بدنبال داشته باشد. اعداد اعشاری تقریبا در تمام کامپیوترها نشان‌دهنده یک مقدار تقریبی هستند.

تبدیل مابین نوع‌های بنیادین بصورت صریح و ضمنی

متغیر **average** بصورت **double** (خط 55 از شکل ۱۳-۴) اعلان شده تا نتیجه اعشاری محاسبه انجام گرفته را در خود ذخیره سازد. با این همه، متغیرهای **total** و **gradeCounter** هر دو از نوع صحیح می‌باشند. بخاطر دارید که نتیجه تقسیم دو عدد صحیح یک عدد صحیح است که در آن بخش اعشاری جواب از بین می‌رود (قطع می‌شود). در عبارت زیر

```
average = total / gradeCounter;
```

ابتدا تقسیم انجام می‌شود، از اینرو بخش اعشاری نتیجه قبل از تخصیص به **average** از بین می‌رود. برای انجام یک محاسبه اعشاری با مقادیر صحیح، بایستی مقادیر موقتی که اعداد اعشاری هستند برای محاسبه ایجاد کنیم. زبان C++ دارای *عملگر غیربااینری cast* است که این وظیفه را انجام می‌دهد. در خط 81 از عملگر cast بصورت **static_cast<double>(total)** برای ایجاد یک کپی موقت اعشاری از عملوند موجود در درون پرانتزها یعنی **total** استفاده شده است. به استفاده از یک عملگر cast به این روش، *تبدیل صریح* می گویند. هنوز مقدار ذخیره شده در **total** یک عدد صحیح است.

اکنون محاسبه متشکل از یک مقدار اعشاری (نسخه **double** موقت از **total**) است که بر یک عدد صحیح در **gradeCounter** تقسیم می‌شود. کامپایلر C++ فقط از نحوه ارزیابی عباراتی که در آن نوع داده‌های عملوندها یکسان هستند، اطلاع دارد. برای اطمینان از اینکه عملوندها از نوع مشابه هستند، کامپایلر مبادرت به انجام عملی بنام *ترفیع* که *تبدیل ضمنی* نیز نامیده می‌شود بر روی عملوندهای انتخابی می کند. برای مثال، در یک عبارت که حاوی مقادیری از نوع داده **int** و **double** است، C++ مبادرت به ترفیع عملوندهای int به مقادیر **double** می کند. در این مثال با **total** همانند یک نوع داده **double** رفتار می کنیم (با استفاده از عملگر cast)، از اینرو کامپایلر مبادرت به ترفیع **gradeCounter** به **double** کرده و به محاسبه اجازه انجام می‌دهد و نتیجه تقسیم اعشاری به **average** تخصیص می‌یابد. در فصل ششم، در مورد نوع داده‌های بنیادین و نحوه ترفیع آنها توضیح خواهیم داد.

خطای برنامه‌نویسی

*می‌توان از عملگر **cast** برای تبدیل مابین نوع‌های بنیادین عددی همانند **int** و **double** و مابین نوع کلاس‌های مرتبط استفاده کرد (در فصل سیزدهم با این موضوع آشنا خواهید شد). تبدیل به یک نوع اشتباه می‌تواند خطای کامپایلر یا خطای زمان اجرا بوجود آورد.*

عملگرهای cast برای استفاده در هر نوع داده و همچنین نوع‌های کلاس در دسترس هستند. بدنبال عملگر **static_cast** یک جفت کاراکتر < و > که نوع داده را احاطه کرده‌اند آورده می‌شود. عملگر cast یک عملگر غیربااینری است. عملگری که فقط یک عملوند اختیار می کند. در فصل دوم، با عملگرهای محاسباتی بااینری آشنا شده‌اید. همچنین C++ از نسخه‌های عملگرهای غیربااینری جمع (+) و منفی (-) پشتیبانی می کند، از اینرو برنامه‌نویس می‌تواند عبارتی مثل 7- یا 5+ بنویسد. عملگر cast از سایر عملگرهای غیربااینری همانند + و – از تقدم بالاتری برخوردار است. این تقدم بالاتر از عملگرهای *، / و % و پایین‌تر از پرانتز است. در جدول شکل ۲۲-۴ این عملگر را با نماد **static_cast<*type*>()** عرضه کرده‌ایم.

قالب‌بندی اعداد اعشاری

قالب‌بندی بکار رفته در برنامه شکل۱۳-۴ را بطور خلاصه در این بخش و بطور دقیق‌تر در فصل پانزدهم توضیح خواهیم داد. فراخوانی تابع **setprecision** در خط 86 (با آرگومان 2) بر این نکته دلالت دارد که متغیر **average** از نوع **double** بایستی با دو رقم معنی‌دار در سمت راست نقطه اعشار چاپ شود (مثلاً 97.37) به اینحالت *کنترل‌کننده جریان پارامتری شده* (استریم) می‌گویند (بدلیل وجود 2 در درون پرانتز). برنامه‌هایی که از این فراخوانی استفاده می‌کنند باید حاوی رهنمود دستور دهنده زیر باشند (خط 10)

```
#include <iomanip>
```

خط 11 تصریح‌کننده نام فایل سرآیند <**iomanip**> است که در این برنامه بکار گرفته خواهد شد. دقت کنید که **endl** یک کنترل‌کننده جریان پارامتری نشده است (چرا که پس از آن مقدار یا عبارتی در درون پرانتزها وجود ندارد) و نیازمند فایل سرآیند <**iomanip**> نیست. اگر دقت تعیین نشود، معمولاً اعداد اعشاری با شش رقم معنی‌دار چاپ می‌شوند (دقت پیش‌فرض در اکثر سیستم‌های 32 بیتی). کنترل‌کننده جریان **fixed** بر این نکته دلالت دارد که مقادیر اعشاری بایستی با خروجی که فرمت نقطه ثابت نامیده می‌شوند چاپ شوند، که متضاد نماد علمی می‌باشد. نماد علمی روشی برای نمایش یک عدد بصورت، عدد اعشاری مابین مقدار1 الی10 است که در توانی از 10 ضرب می‌شود. برای مثال، مقدار3100 را می‌توان در نماد علمی بصورت 3.1×10^3 به نمایش در آورد. به هنگام نمایش مقادیری که بسیار بزرگ یا بسیار کوچک هستند، نماد علمی می‌تواند ابزار مناسبی برای اینکار باشد در فصل پانزدهم با قالب‌بندی نماد علمی آشنا خواهید شد. در طرف مقابل، قالب‌بندی نقطه ثابت قرار دارد که یک عدد اعشاری را مجبور می‌کند تا به تعداد مشخص شده مبادرت به نمایش ارقام کند. همچنین این فرمت نقطه اعشار و دنباله صفرها در چاپ را کنترل می‌کند، حتی اگر عدد یک عدد صحیح باشد، همانند 88.00، بدون قالب‌بندی نقطه ثابت چنین عددی در C++ بصورت 88 چاپ می‌شود، بدون دنباله صفرها و نقطه اعشار. زمانیکه از کنترل‌کننده‌های جریان **fixed** و **setprecision** در برنامه‌ای استفاده می‌شود، مقادیر چاپ شده به تعداد نقاط دیسمال که توسط مقدار ارسالی به **setprecision** مشخص می‌شود، گرد می‌شوند (همانند مقدار 2 در خط 86)، اگرچه مقدار موجود در حافظه بدون تغییر باقی می‌ماند. برای مثال، مقادیر 87.946 و 67.543 بصورت 87.95 و 67.54 چاپ می‌شوند. توجه کنید که می‌توان نقطه اعشار را با استفاده از کنترل‌کننده جریان **showpoint** به نمایش درآورد. اگر **showpoint** بدون **fixed** بکار گرفته شود، دنباله صفحه‌ها چاپ نخواهد شد. همانند **endl**، کنترل‌کننده‌های جریان**fixed** و**showpoint**

پارامتری شده نبوده و نیازی به سرآیند فایل <**iomanip**> ندارند. هر دو آنها را می‌توان در سرآیند <**iostream**> پیدا کرد.

خط 86 و 87 از شکل ۱۳-۴ خروجی میانگین کلاس هستند. در این مثال میانگین کلاس گرد شده به نزدیکترین صدم و دقیقاً با دو رقم در سمت راست نقطه اعشار به نمایش درآمده‌اند. کنترل‌کننده جریان پارامتری شده (خط 86) نشان می‌دهد که مقدار متغیر **average** بایستی با دقت دو رقم در سمت راست نقطه اعشار به نمایش درآید (**setprecision(2)**). در اجرای نمونه‌ای برنامه سه نمره وارد برنامه ۱۴-۴ شده که مجموع آنها 257 شده است و میانگین حاصل از این رقم عدد 85.666666 است. کنترل‌کننده جریان پارامتری شده **setprecision** سبب می‌شود تا مقدار به تعداد رقم مشخص گرد شود. در این برنامه، میانگین به 85.67 گرد شده است.

۱۰-٤ فرموله کردن الگوریتم‌ها: عبارات کنترلی تودرتو

اجازه دهید تا به بررسی مسئله دیگری بپردازیم. مجدداً الگوریتم را با استفاده از شبه‌کد و از بالا به پایین، اصلاح گام به گام فرموله کرده و سپس برنامه C++ مربوط به آنرا خواهیم نوشت. در مثال‌های قبلی مشاهده کردید که عبارتهای کنترلی همانند یک پشته یکی بر روی دیگری و به ترتیب قرار داده می‌شدند. در این مرحله، به معرفی روشی خواهیم پرداخت که عبارتهای کنترلی در آن را می‌توان با یکدیگر ترکیب کرد، بطوریکه عبارتی در درون عبارت دیگر جای می‌گیرد.

به صورت مسئله توجه نمائید:

یک کالج با برگزاری دوره‌ای دانشجویان را آماده امتحان پایان ترم می‌کند. سال گذشته، ۱۰ تن از دانشجویان که این دوره را گذرانده بودند در امتحان پایان ترم شرکت کردند. مدیریت کالج می‌خواهد از وضعیت دانشجویان شرکت کرده در امتحان مطلع شود. از شما خواسته شده تا برنامه‌ای بنویسید تا خلاصه‌ای از نتایج آزمون ارائه دهد. لیستی از ۱۰ دانشجو دریافت کرده و سپس در کنار نام کسانی که در آزمون قبول شده‌اند 1 و کسانی که در آزمون مردود شده‌اند 2 چاپ شود.

این برنامه باید بصورت زیر نتایج آزمون را تحلیل نماید:

۱- وارد کردن نتیجه هر آزمون (برای مثال 1 یا 2). نمایش پیغام "Enter result" در هر بار که برنامه درخواست نتیجه آزمون می‌کند.

۲- شمارش تعداد قبولی‌ها و مردودی‌ها.

۳- نمایش خلاصه‌ای از نتایج آزمون، شامل تعداد دانشجویان که موفق به گذراندن آزمون شده‌اند و تعدادی که مردود شده‌اند.

۴-اگر بیش از ۸ دانشجو از آزمون با موفقیت عبور کرده‌اند. پیغام "Raise tuition" به نمایش درآید.

پس از مطالعه صورت مسئله، تصمیمات زیر را برای حل آن اتخاذ می‌کنیم:

۱- برنامه باید بر روی نتایج آزمون 10 دانشجو کار کند، از اینرو حلقه شمارنده - کنترل می‌تواند بکار گرفته شود.

۲- نتیجه هر آزمون عدد 1 یا 2 است. هر بار که برنامه اقدام به خواندن نتیجه یک آزمون می‌کند، برنامه باید یک 1 یا 2 دریافت نماید.

۳- دو شمارنده به ذخیره‌سازی نتایج آزمون می‌پردازند. یکی برای شمارش تعداد دانشجویان که از آزمون با موفقیت عبور کرده‌اند و دیگری برای شمارش تعدادی که در آزمون مردود شده‌اند.

۴- پس از اینکه برنامه تمام نتایج را مورد پردازش قرار داد، باید تعیین کند که آیا تعداد قبولی‌ها بیش از هشت نفر است یا خیر.

اجازه دهید تا با روش از بالا به پایین، اصلاح گام‌به‌گام کار را دنبال کنیم. عبارت شبه‌کد زیر در بالاترین سطح (top) قرار دارد:

Analyze exam result and decide if tuition should be raised

مجدداً یادآوری می‌کنیم که عبارت top توصیف کلی در مورد برنامه است و قبل از اینکه بتوان شبه‌کد را به فرم یک برنامه C++ نوشت انجام چندین مرحله اصلاح گام‌به‌گام ضروری است. اولین اصلاح عبارت است از:

Initialize variables
Input the 10 exam grades and count passes and failures
Print a summary of the exam result and decide if tuition should be raised

حتی زمانیکه یک تصور کامل از کل برنامه بدست آورده باشیم، انجام اصلاحات بعدی مورد نیاز است. باید به دقت به بررسی و مشخص کردن متغیرها پرداخت. شمارنده‌ها به منظور ثبت قبولی‌ها و مردودی‌ها مورد نیاز هستند. یک شمارنده، کنترل کننده حلقه بوده و یک متغیر، ورودی کاربر را ذخیره می‌کند. عبارت شبه‌کد زیر

Initialize variables

می‌تواند بصورت زیر اصلاح شود:

Initialize passes to zero
Initialize failures to zero
Initialize student counter to one

فقط شمارنده‌های، تعداد قبولی‌ها و مردودی‌ها و تعداد دانش‌آموزان مقداردهی اولیه می‌شود. عبارت شبه‌کد

Input the 10 quiz grades and count passes and failures

مستلزم یک حلقه برای وارد کردن نتیجه هر آزمون است. در این برنامه بدلیل اینکه از همان ابتدا تعداد نتایج آزمون مشخص است (۱۰)، از اینرو می‌توان از روش شمارنده-کنترل تکرار استفاده کرد. در درون

حلقه یک عبارت انتخابی دو گانه تعیین می‌کند که نتیجه آزمون قبولی است یا مردودی و شمارنده مربوطه یک واحد افزایش می‌یابد. اصلاح عبارت شبه‌کد قبلی می‌تواند بصورت زیر انجام شود:

```
While student counter is less than or equal to 10
    Prompt the user to enter the next exam result
    Input the next exam result
    If the student passed
            Add one to passes
    Else
            Add one to failures
    Add one to student counter
```

به نحوه استفاده از خطوط خالی در میان مجموعه ***if..else*** دقت کنید که باعث افزایش خوانایی برنامه شده است. عبارت شبه‌کد

```
Print a summary of the exam results and decide if tuition should be raised
```

می‌تواند بصورت زیر اصلاح شود:

```
Print the number of passes
Print the number of failures
If more than eight student passed
    Print "Raise tuition"
```

در شکل ۱۵-۴ دومین مرحله اصلاح بصورت کامل نشان داده شده است. به کاربرد خطوط خالی در میان ساختار **while** توجه کنید، که باعث افزایش خوانائی برنامه می‌شوند. اکنون این شبه‌کد بقدر کافی برای تبدیل به یک برنامه C++ آماده شده است.

```
Initialize passes to zero
Initialize failures to zero
Initialize student to zero
While student counter is less than or equal to ten
        Input the next exam result
        If the student passed
                Add one to passes
        Else
                Add one to failures
                Add one to student counter
Print the number of passes
Print the number of failures
If more than eight students passed
        Print "Raise tuition"
```

شکل ۱۵-٤ | شبه‌کد برنامه نتیجه آزمون.

تبدیل به کلاس Analysis

کلاس C++ که مبادرت به پیاده‌سازی الگوریتم شبه‌کد کرده است در شکل‌های ۱۶-۴ و ۱۷-۴ و دو اجرای نمونه در شکل ۱۸-۴ نشان داده است.

```
1  // Fig. 4.16: Analysis.h
2  // Definition of class Analysis that analyzes examination results.
3  // Member function is defined in Analysis.cpp
4
5  // Analysis class definition
6  class Analysis
7  {
8  public:
9     void processExamResults(); // process 10 students' examination results
10 }; // end class Analysis
```

شکل ١٦-٤ | برنامه بررسی نتیجه آزمون: فایل سرآیند Analysis.

```
1  // Fig. 4.17: Analysis.cpp
2  // Member-function definitions for class Analysis that
3  // analyzes examination results.
4  #include <iostream>
5  using std::cout;
6  using std::cin;
7  using std::endl;
8
9  // include definition of class Analysis from Analysis.h
10 #include "Analysis.h"
11
12 // process the examination results of 10 students
13 void Analysis::processExamResults()
14 {
15    // initializing variables in declarations
16    int passes = 0; // number of passes
17    int failures = 0; // number of failures
18    int studentCounter = 1; // student counter
19    int result; // one exam result (1 = pass, 2 = fail)
20
21    // process 10 students using counter-controlled loop
22    while ( studentCounter <= 10 )
23    {
24       // prompt user for input and obtain value from user
25       cout << "Enter result (1 = pass, 2 = fail): ";
26       cin >> result; // input result
27
28       // if...else nested in while
29       if ( result == 1 )          // if result is 1,
30          passes = passes + 1;     // increment passes;
31       else                        // else result is not 1, so
32          failures = failures + 1; // increment failures
33
34       // increment studentCounter so loop eventually terminates
35       studentCounter = studentCounter + 1;
36    } // end while
37
38    // termination phase; display number of passes and failures
39    cout << "Passed " << passes << "\nFailed " << failures << endl;
40
41    // determine whether more than eight students passed
```

```
42    if ( passes > 8 )
43       cout << "Raise tuition " << endl;
44 } // end function processExamResults
```

شکل ۱۷-٤ |برنامه بررسی نتیجه آزمون: عبارات کنترلی تودرتو در فایل کد منبع Analysis

```
1  // Fig. 4.18: fig04_18.cpp
2  // Test program for class Analysis.
3  #include "Analysis.h" // include definition of class Analysis
4
5  int main()
6  {
7     Analysis application; // create Analysis object
8     application.processExamResults(); // call function to process results
9     return 0; // indicate successful termination
10 } // end main
```

```
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Passed: 9
Failed: 1
Raise Tuition
```

```
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Passed: 6
Failed: 4
```

شکل ۱۸-٤ | برنامه تست کننده کلاس Analysis

خطوط 18-16 از برنامه۱۷-۴ متغیرهای اعلان کرده‌اند که تابع عضو **processExamResults** از کلاس **Analysis** از آنها برای پردازش نتایج آزمون استفاده می‌کند. توجه کنید که از یکی از ویژگی‌های زبان C++ استفاده کرده‌ایم که به مقداردهی اولیه متغیر امکان می‌دهد تا با بخش اعلان یکی شود (**passes** با صفر، **failures** با صفر و **studentCounter** با 1 مقداردهی اولیه شده‌اند). امکان مقداردهی در ابتدای تکرار هر حلقه وجود دارد، معمولاً چنین مقداردهی‌های مجددی توسط عبارات تخصیصی بجای اعلان‌ها یا انتقال اعلان‌ها بدرون بدنه حلقه صورت می‌گیرند.

حلقه while ده بار تکرار می‌شود (خطوط 36-22). در هر تکرار، حلقه یک نتیجه آزمون دریافت و آن را پردازش می‌کند. دقت کنید که عبارت **if..else** در خطوط 32-29 برای پردازش هر نتیجه در عبارت **while** بصورت تودرتو قرار گرفته است. اگر **result** برابر **1** باشد، عبارت **if..else** یک واحد به **passes** اضافه می‌کند و در غیر اینصورت فرض می‌کند که **result** برابر **2** بوده و یک واحد به **failures** اضافه می‌نماید. خط 35 مبادرت به افزایش **studentCouunter** قبل از اینکه شرط تست حلقه در خط 22 صورت گیرد می‌کند. پس دریافت 10 مقدار، حلقه پایان یافته و خط 39 تعداد قبولی‌ها (**passes**) و مردودی‌ها (**failures**) را به نمایش در می‌آورد. عبارت **if** در خطوط 43-42 تعیین می‌کند که آیا تعداد دانشجویان قبول شده بیش از هشت نفر است یا خیر. اگر چنین باشد پیغام "Raise Tuition" به نمایش در می‌آید.

بررسی کلاس Analysis

در برنامه ۱۸-۴ یک شی **Analysis** ایجاد (خط 7) و تابع عضو **processExamResults** فراخوانی می‌شود (خط 8) تا مجموع نتایج آزمون وارد شده توسط کاربر پردازش شود. دو اجرای نمونه از برنامه ۱۸-۴ در خروجی نشان داده شده است. در انتهای اولین اجرا، شرط موجود در خط 42 تابع عضو **procesExamResults** در شکل ۱۷-۴ برقرار شده (بیش از هشت دانشجو قبول شده‌اند) و از اینرو پیغام "Raise Tuition" به نمایش درآمده است.

۱۱-٤ عملگرهای تخصیص‌دهنده

C++ دارای چندین عملگر تخصیص‌دهنده برای کاستن از طول عبارات تخصیصی است. برای مثال، عبارت

```
c = c + 3;
```

می‌تواند بصورت زیر و با استفاده از عملگر =+ نوشته شود

```
c += 3;
```

عملگر =+ مقدار عملوند قرار گرفته در سمت راست را به مقدار عملوند سمت چپ اضافه کرده و نتیجه آنرا در متغیر عملوند سمت چپ ذخیره می‌کند. هر عبارتی به فرم زیر را

متغیر = عبارت عملگر متغیر;

می‌توان به فرم زیر نوشت:

متغیر عملگر = عبارت;

عملگر یکی از عملگرهای باینری +، -، *، / یا % و متغیر یک مقدار سمت چپ (lvalue) است. مقدار سمت چپ، متغیری است که در سمت چپ یک عبارت تخصیصی جای می‌گیرد. جدول شکل ۱۹-۴ حاوی عملگرهای تخصیص دهنده محاسباتی و عبارات نمونه‌ای است که از این عملگرها استفاده می‌کنند.

عملگر تخصیص دهنده	عبارت نمونه	معادل	تخصیص
با فرض c = 3، d=5، e=4، f=6 و g=12			

c به 10	c = c + 7	c += 7	+=
d به 1	d = d - 4	d -= 4	-=
e به 20	e = e * 5	e *= 5	*=
f به 2	f = f / 2	f /= 3	/=
g به 3	g = g % 9	g %= 9	%=

شکل ۱۹-٤ | عملگرهای تخصیص‌دهنده.

۱۲-٤ عملگرهای افزاینده و کاهنده

علاوه بر عملگرهای محاسباتی تخصیص دهنده، زبان C++ دارای *عملگر افزاینده* غیربااینری ++ و *عملگر کاهنده* غیربااینری -- است. این عملگرها در جدول شکل ۲۰-۴ توضیح داده شده‌اند. اگر متغیر **c** بخواهد یک واحد افزایش پیدا کند، عملگر افزاینده ++ را می‌توان بجای استفاده از عبارت **c = c + 1** یا **c += 1** بکار گرفت. اگر عملگر افزاینده یا کاهنده قبل از یک متغیر قرار داده شود، به مفهوم *پیش‌افزایش یافته* یا *پیش‌کاهش یافته* خواهد بود. اگر عملگر افزاینده یا کاهنده پس از یک متغیر بکار گرفته شود، به مفهوم *پس‌افزایشی* یا *پس‌کاهشی* خواهد بود.در هر دو حالت افزایشی یا کاهشی مقدار متغیر یک واحد افزایش یا کاهش پیدا می‌کند. پس از انجام کار، مقدار جدید متغیر در عبارتی که حاوی آن است بکار گرفته می‌شود.

عملگر	عبارت نمونه	عنوان	توضیح
++	++a	پیش‌افزایشی	**a** یک واحد افزایش می‌یابد، سپس مقدار جدید **a** در عبارتی که حاوی **a** است بکار گرفته می‌شود.
++	a++	پس‌افزایشی	از مقدار جاری **a** در عبارتی که حاوی آن است استفاده شده، سپس مقدار **a** یک واحد افزایش میابد.
--	--b	پیش‌کاهشی	**b** یک واحد کاهش می‌یابد، سپس مقدار جدید **b** در عبارتی که حاوی **b** است بکار گرفته می‌شود.
--	b--	پس‌کاهشی	از مقدار جاری **b** در عبارتی که حاوی آن است استفاده شده، سپس مقدار **b** یک واحد کاهش میابد.

شکل ۲۰-٤ | عملگرهای افزاینده و کاهنده.

برنامه شکل۲۱-۴ به توصیف تفاوت موجود مابین نسخه‌های پیش‌افزایش و پس‌افزایش عملگر افزاینده ++ می‌پردازد. عملگر کاهنده -- نیز به طریق مشابهی کار می‌کند. توجه کنید که این مثال حاوی کلاس نمی‌باشد، اما فایل کد منبع با **main** تمام برنامه‌ها کار می‌کند. در این فصل و فصل سوم شاهد مثال‌های بوده‌اید که حاوی یک کلاس (شامل سرآیند و فایلهای کد منبع برای این کلاس بوده‌اند) به همراه فایل کد منبع دیگری برای تست کلاس بودند. این فایل کد منبع حاوی تابع **main** است که یک شی از کلاس ایجاد و توابع عضو خود را فراخوانی می‌کند. در این مثال، فقط خواسته‌ایم تا مکانیزم عملگر++ را به نمایش در آوریم، از اینرو فقط از یک فایل کد منبع با تابع **main** استفاده کرده‌ایم. خط 12 مبادرت به

مقداردهی اولیه متغیر c با 5 و خط 13 مقدار اولیه c را به نمایش در می‌آورد. خط 14 مقدار عبارت c++ را چاپ می‌کند. این عبارت سبب پس‌افزایش متغیر c شده و در نتیجه مقدار اولیه c یعنی 5 چاپ می‌گردد، سپس مقدار c افزایش می‌یابد. از اینرو، خط 14 مقدار اولیه c یعنی 5 را مجدداً چاپ می‌کند. خط 15 مقدار جدید c یعنی 6 را برای تاکید بر این نکته که مقدار متغیر براستی در خط 14 افزایش یافته است چاپ می‌نماید.

```
1  // Fig. 4.21: fig04_21.cpp
2  // Preincrementing and postincrementing.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     int c;
10
11    // demonstrate postincrement
12    c = 5; // assign 5 to c
13    cout << c << endl; // print 5
14    cout << c++ << endl; // print 5 then postincrement
15    cout << c << endl; // print 6
16
17    cout << endl; // skip a line
18
19    // demonstrate preincrement
20    c = 5; // assign 5 to c
21    cout << c << endl; // print 5
22    cout << ++c << endl; // preincrement then print 6
23    cout << c << endl; // print 6
24    return 0; // indicate successful termination
25 } // end main
```

```
5
5
6

5
6
6
```

شکل ۲۱-٤ |تفاوت مابین عملگرهای پیش‌افزایشی و پس‌افزایشی.

خط 20 مقدار متغیر c را به 5 باز می‌گرداند و خط 21 مقدار c را چاپ می‌کند. خط 22 مبادرت به چاپ مقدار عبارت c++ را می‌کند. این عبارت سبب پیش‌افزایش c شده است، از اینرو مقدار آن افزایش یافته و سپس مقدار جدید یعنی 6 چاپ می‌شود. خط 23 مجدداً مقدار c را به نمایش در می‌آورد تا نشان دهد که مقدار c هنوز پس از اجرای خط 22 برابر 6 است.

عملگرهای تخصیص ریاضی و عملگرهای افزاینده و کاهنده می‌توانند عبارات برنامه‌نویسی را ساده‌تر کنند. سه عبارت تخصیصی در برنامه ۱۷-۴

```
passes = passes + 1;
failures = failures + 1;
studentCounter = studentCounter + 1;
```

را می‌توان با استفاده از عملگرهای تخصیصی بصورت زیر هم نوشت

```
passes += 1;
failures += 1;
student += 1;
```

با عملگرهای پیش‌افزایشی بصورت زیر

```
++passes;
++failures;
++studentCounter;
```

با عملگرهای پس‌افزایشی بصورت زیر نوشت

```
passes++
failures++;
studentCounter++;
```

خطای برنامه‌نویسی

مبادرت به استفاده از عملگر افزاینده یا کاهنده بر روی عبارتی بجز نام یک متغیر، همانند (x + 1)++ خطای نحوی خواهد بود.

جدول شکل ۲۲-۴ نمایشی از تقدم و ارتباط عملگرهای مطرح شده تا بدین‌جا را عرضه کرده است. نمایش عملگرها با تقدم آنها از بالا به پایین است. ستون دوم توصیف کننده ارتباط عملگرها در هر سطح تقدم است. دقت کنید که عملگر شرطی (?:)، عملگر غیربااینری پس‌افزایشی (++)، پس‌کاهشی (--) ، جمع (+)، تفریق (-)، و عملگرهای = ، =+ ، =* ، =/ و =% از چپ به راست ارزشیابی می‌شوند. مابقی عملگرهای جدول شکل ۲۲-۴ از راست به چپ می‌باشند. ستون سوم اسامی عملگرها را نشان می‌دهد.

عملگر	ارتباط	نوع
( )	چپ به راست	پرانتز
++ -- static_cast< *type* >()	چپ به راست	غیربااینری
++ -- + -	راست به چپ	غیربااینری
* / %	چپ به راست	تعددی
+ -	چپ به راست	افزاینده کاهنده
<< >>	چپ به راست	درج/استخراج
< <= > >=	چپ به راست	رابطه‌ای
== !=	چپ به راست	برابری
?:	راست به چپ	شرطی
= += -= *= /= %=	راست به چپ	تخصیصی

شکل ۲۲-٤ |تقدم و رابطه عملگرهای مطرح شده تا این مرحله.

۱۳-٤ مبحث آموزشی مهندسی نرم‌افزار: شناسایی صفات کلاس در سیستم ATM

در بخش ۱۱-۳ اولین مرحله از طراحی شی‌گرا (OOD) را برای سی‌ستم ATM انجام دادیم. تحلیل مستند نیازها و شناسایی کلاس‌های مورد نیاز در پیاده‌سازی سیستم. همچنین اسامی و کلمات کلیدی موجود در مستند نیازها را مشخص و کلاس‌ها را مجزا کرده و نقشی که هر یک در سیستم ATM بازی می‌کنند را شناسایی کردیم. سپس کلاس‌ها و روابط آنها را توسط دیاگرام کلاس UML مدل‌سازی کردیم (شکل ۲۳-۳). کلاس‌ها دارای صفات (داده) و عملیات (رفتار) هستند. صفات کلاس در برنامه‌های C++ بعنوان عضوهای داده و عملیات کلاس توسط توابع عضو پیاده‌سازی می‌شوند. در این بخش، به تعیین برخی از صفات مورد نیاز در سیستم ATM می‌پردازیم. در فصل پنجم، بررسی می‌کنیم چگونه این صفات در تعیین وضعیت یک شی نقش دارند. در فصل ششم، به تعیین عملیات کلاس‌ها خواهیم پرداخت.

شناسایی صفات

به صفات برخی از شی‌ها در جهان واقعی توجه کنید: هر فردی دارای قد، وزن بوده و می‌تواند چپ‌دست، راست‌دست یا قادر به نوشتن با هر دو دست باشد. صفات یک رادیو شامل تنظیم ایستگاه، تنظیم صدا و تنظیمات AM یا FM است. صفات یک اتومبیل شامل دور موتور، حجم مخزن سوخت و نوع جعبه‌دنده است. صفات یک کامپیوتر شخصی شامل سازنده آن (همانند Dell، Sun، Apple یا IBM)، نوع صفحه نمایش (مثلاً LCD یا CRT)، میزان حافظه اصلی و سایز دیسک سخت است.

می‌توانیم صفات بسیار زیادی برای کلاس‌ها را در سیستم با دقت در کلمات توصیف‌کننده و عبارات موجود در مستند نیازها پیدا کنیم. برای هر صفتی که نقشی در سیستم بازی می‌کند یک صفت ایجاد کرده و آن را به یک یا چند کلاس شناسایی شده در بخش ۱۱-۳ تخصیص می‌دهیم. همچنین صفات را برای نمایش داده‌های اضافی که ممکن است کلاس نیاز داشته باشد یا داده‌های که می‌توانند فرآیند طراحی را مشخص‌تر سازند، ایجاد می‌کنیم.

در جدول شکل ۲۳-۴ کلمات و عبارت بدست آمده از مستند نیازها را که توصیف‌کننده کلاس هستند، لیست شده‌اند. ترتیب دستیابی به این کلاس بر اساس ظاهر شدن آنها در مستند نیازها است.

کلاس	کلمات و جملات توصیفی
ATM	تایید هویت کاربر
BalanceInquiry	شماره حساب
Withdrawl	شماره حساب

	موجودی
Deposit	شماره حساب
	موجودی
BankDatabase	[کلمه یا جمله توصیف‌کننده وجود ندارد]
Account	شماره حساب
	PIN
	مانده حساب
Screen	[کلمه یا جمله توصیف‌کننده وجود ندارد.]
keypad	[کلمه یا جمله توصیف‌کننده وجود ندارد.]
CashDispenser	هر روز با 500 عدد اسکناس 20 دلاری شروع بکار می‌کند.
DepositeSlot	[کلمه یا جمله توصیف‌کننده وجود ندارد.]

شکل ۲۳-٤ | کلمات و عبارات توصیف‌کننده از مستند نیازهای ATM.

جدول ۲۳-۴ ما را به سمت ایجاد یک صفت کلاس ATM سوق می‌دهد. کلاس ATM مسئول نگهداری اطلاعاتی در ارتباط با وضعیت ATM است. عبارت «تایید هویت کاربر» توصیف‌کننده وضعیتی از ATM است (در بخش۱۱-۵ به معرفی وضعیت‌ها خواهیم پرداخت)، از اینرو **userAuthenticated** را بعنوان یک صفت بولی در نظر گرفته‌ایم (صفتی که دارای یک مقدار true یا **fasle** است). نوع **Boolean** در UML معادل نوع **bool** در زبان C++ است. این صفت بر این نکته دلالت دارد که آیا ATM با موفقیت هویت کاربر جاری را تایید کرده است یا خیر، برای اینکه سیستم به کاربر اجازه انجام تراکنش و دسترسی به اطلاعات حساب را فراهم آورد بایستی ابتدا **userAuthenticated** برابر **true** باشد. این صفت در حفظ امنیت داده‌ها در سیستم نقش مهمی ایفا می‌کند.

کلاس‌های **BalanceInquiry**، **Withdrawal** و **Deposit** یک صفت را به اشتراک می‌گذارند. هر تراکنشی مستلزم یک «شماره‌حساب» است که متناظر با حساب کاربری است که تراکنش را انجام می‌دهد. یک صفت صحیح **accountNumber** به هر کلاس تراکنش برای شناسایی حساب اهدا می‌کنیم.

کلمات و جملات توصیفی در مستند نیازها پیشنهاد برخی از صفات متفاوت و مورد نیاز برای هر کلاس ترکنش را می‌کنند. مستند نیازها بر این نکته دلالت دارد که برای برداشت پول نقد یا سپرده‌گذاری، باید کاربر یک «مقدار» مشخص از پول را برای برداشت یا سپرده‌گذاری مشخص کند. از اینرو، به کلاس‌های **Withedrawal** و **Deposit** یک صفت بنام **amount** تخصیص می‌دهیم تا مقدار مشخص شده از سوی کاربر را در خود ذخیره سازد. میزان پول مرتبط با برداشت پول و سپرده‌گذاری، تعریف‌کننده مشخصه این تراکنش‌ها است که سیستم نیازمند آنها می‌باشد. کلاس **BalanceInquiry** نیازی به داده‌های اضافی برای انجام وظیفه خود ندارد. تنها نیاز این کلاس یک شماره‌حساب است تا براساس آن موجودی حساب را بازیابی کند.

کلاس **Account** دارای چندین صفت است. مستند نیازها مشخص می‌کند که هر حساب بانکی دارای یک «شماره‌حساب» و "PIN" است، که سیستم با استفاده از آن مبادرت به شناسایی حساب و هویت کاربران می‌کند. به کلاس **Account** دو صفت صحیح تخصیص داده‌ایم: **PIN** و **accoutnNumber**. همچنین مستند نیازها تصریح می‌کند که هر حسابی مبادرت به نگهداری «موجودی» از میزان پولی که در حساب وجود دارد و مقدار پولی که کاربر بعنوان سپرده در پاکت وارد سیستم ATM وارد کرده ولی هنوز توسط مامور بانک تایید نشده و چک‌هایی که وارد حساب نشده‌اند، می‌کند. با این همه، باید حساب، میزان موجودی که کاربر سپرده‌گذاری کرده است را ثبت کند. بنابر این، تصمیم گرفته‌ایم که یک حساب باید قادر به نمایش میزان موجودی با استفاده از دو صفت UML از نوع **Double** باشد: **availableBalance** و **totalBalance**. صفت **availableBalance** تعیین‌کننده میزان پولی است که کاربر می‌تواند بصورت نقد از حساب خود برداشت کند. صفت **totalBalance** به کل موجودی اشاره دارد که شامل سپرده‌گذاری نیز می‌شود. برای مثال، فرض کنید یک کاربر ATM مبلغ 50.00 دلار در یک حساب خالی سپرده‌گذاری کرده است. در اینحالت صفت **totalBalance** به 50.00 دلار افزایش می‌یابد تا میزان سپرده ثبت گردد، اما مقدار صفت **availableBalance** هنوز در صفر دلار باقی می‌ماند.

کلاس **CashDispenser** دارای یک صفت است. مستند نیازها مشخص می‌کند که تحویل‌دار خودکار «هر روز کار خود را با 500 قطعه اسکناس 20 دلاری شروع می‌کند.» این تحویل‌دار باید مراقب تعداد اسکناس‌ها موجود باشد تا بتواند تعیین کند که آیا به میزان کافی اسکناس برای پرداخت به تقاضای صورت گرفته در اختیار دارد یا خیر. به کلاس **CashDispenser** یک صفت صحیح به نام **count** تخصیص می‌دهیم. که در ابتدای کار با 500 تنظیم شده است.

در برنامه‌های واقعی هیچ تضمینی وجود ندارد که مستند نیازها به قدر کافی غنی، دقیق و گویا برای طراحان سیستم‌های شی‌گرا باشد تا آنها هم بتوانند تمام صفات یا حتی تمام کلاس‌ها را تعیین کنند. نیاز به کلاس‌ها، صفات و رفتارهای اضافی در فرآیند طراحی خود را آشکار می‌کنند. همانطوری که در این مبحث آموزشی پیش می‌رویم، مبادرت به افزودن، اصلاح و حذف اطلاعاتی در ارتباط با کلاس‌ها در سیستم خود خواهیم کرد.

مدل کردن صفات

در دیاگرام کلاس شکل ۲۴-۴ برخی از صفات کلاس‌های موجود در سیستم ATM به نمایش درآمده‌اند. جدول شکل ۲۳-۴ در شناسایی این صفات به ما کمک کرده است. برای سادگی کار، شکل ۲۴-۴ نمایشگر وابستگی موجود مابین کلاس‌ها نیست که آنها را در شکل ۲۳-۳ قبلاً عرضه کرده بودیم. از بخش‌های قبل بخاطر دارید که در UML، صفات کلاس در بخش میانی دیاگرام کلاس قرار داده می‌شوند. نام هر صفت و نوع آن توسط یک کولن از هم جدا شده و سپس در برخی از موارد یک علامت تساوی و مقدار اولیه هم بعد از آنها آورده می‌شود.

به صفت **userAuthentiated** از کلاس ATM توجه کنید:

```
userAuthenticated : Boolean = false
```

در اعلان این صفت سه نوع داده در ارتباط با آن وجود دارد. نام صفت **userAuthenticated** است. نوع صفت **Boolean** است. در C++، یک صفت را می‌توان توسط یک نوع بنیادین همانند **bool**، **int** یا **double** یا یک نوع کلاس عرضه کرد. در شکل ۲۴-۴ از نوع‌های بنیادین برای صفات استفاده کرده‌ایم. همچنین مقدار اولیه صفت را هم مشخص کرده‌ایم. صفت **userAuthenticated** در کلاس **ATM** دارای مقدار اولیه **false** است. به این معنی که سیستم در ابتدای کار کاربر را تایید نمی‌کند. اگر صفتی دارای مقدار اولیه مشخص شده‌ای نباشد، فقط نام و نوع صفت به نمایش در می‌آیند. برای مثال صفت **accountNumber** از کلاس **BalanceInquiry** از نوع **Integer** است که مقدار اولیه هم ندارد چرا که مقدار این صفت عددی است که هنوز از آن اطلاعی نداریم. این عدد در زمان اجرای برنامه و براساس شماره حساب وارد شده توسط کاربر جاری ATM تعیین می‌شود.

شکل ٢٤-٤ | کلاس‌ها همراه با صفات.

شکل ۲۴-۴ حاوی صفاتی برای کلاس‌های **Screen**، **Keypad** و **DepositSlot** نیست. این اجزاء جزء کامپونت‌های مهم در سیستم هستند که هنوز طراحی ما قادر به تعیین صفات آنها نشده است. با این

وجود در ادامه روند طراحی یا به هنگام پیاده‌سازی این کلاس‌ها در C++ می‌توان به بررسی آنها پرداخت. چنین حالتی در فرآیند مهندسی نرم‌افزار کاملاً طبیعی است.

مهندسی نرم‌افزار

در مراحل اولیه فرآیند طراحی، برخی از کلاس‌ها فاقد صفات (و عملیات) هستند. با این وجود، چنین کلاس‌هایی نباید از نظر دور نگه داشته شوند، چرا که این صفات (و عملیات) می‌توانند خود را در مراحل بعدی طراحی و پیاده‌سازی نشان دهند.

همچنین توجه کنید که در شکل ۲۴-۴ صفتی برای کلاس **BankDatabase** در نظر گرفته نشده است. از فصل سوم بخاطر دارید که در C++، صفات را می‌توان با نوع‌های بنیادین یا نوع‌های کلاس عرضه کرد. چون در مدل کردن دیاگرام‌های کلاس این شکل تصمیم بر استفاده از نوع‌های بنیادین برای صفات گرفته‌ایم، این کلاس فعلاً صفتی ندارد. مدل کردن صفت از نوع کلاس بسیار واضح بوده و همانند یک رابطه (در عمل یک ترکیب) مابین کلاس با صفت است. برای مثال، دیاگرام کلاس در شکل ۲۳-۳ نشان می‌دهد که کلاس **BankDatabase** در یک رابطه ترکیبی با صفر یا چند شی **Account** شرکت دارد. از این ترکیب، می‌توانیم تعیین کنیم که به هنگام پیاده‌سازی سیستم ATM در C++، ملزم به ایجاد صفتی از کلاس **BankDatabase** هستیم که صفر یا بیشتر شی Acount در خود نگهداری کند. به همین ترتیب با کلاس‌های **Screen**، **Keypad**، **CashDispenser** و **DepositSlot** رفتار می‌کنیم. مدل کردن این صفات مبتنی بر ترکیب می‌تواند سبب افزونگی در شکل۲۴-۴ شود، چرا که ترکیب‌های مدل شده در شکل ۲۳-۳ بر این واقعیت تاکید دارند که پایگاه داده حاوی اطلاعاتی در ارتباط با صفر یا بیشتر حساب بوده و ATM مرکب از صفحه‌نمایش،صفحه‌کلید، پرداخت‌کننده‌پول و شکاف‌سپرده‌گذاری است. معمولاً طراحان نرم‌افزاری چنین روابط کامل/ بخش را بصورت ترکیبی بجای صفات مدل‌سازی می‌کنند.

دیاگرام کلاس در شکل ۲۴-۴ ساختار پایه‌ای مدل ما را نشان می‌دهد، اما کامل نیست. در بخش ۱۱-۵ مبادرت به شناسایی وضعیت و فعالیت شی‌ها در مدل کرده و در بخش ۲۲-۶ به بررسی عملیاتی که شی‌ها انجام می‌دهند می‌پردزایم.

تمرینات خودآزمایی مبحث مهندسی نرم‌افزار

۱-۴ عموماً شناسایی صفات کلاس‌ها در سیستم با تحلیل _____در مستند نیازها صورت می‌گیرد.

a) اسامی و جملات

b) کلمات و جملات توصیفی

c) افعال

d) همه گزینه‌های فوق

۴-۲ کدامیک از موارد زیر نشاندهنده صفتی از یک هواپیما نیستند؟

a) طول

b) طول بال هواپیما

c) پرواز

d) تعداد صندلی‌ها

۴-۳ به توضیح مفهوم اعلان صفت کلاس **CashDispenser** در دیاگرام کلاس شکل ۴-۲۴ بپردازید:

```
count : Integer = 500
```

پاسخ خودآزمایی مبحث آموزشی مهندسی نرم‌افزار

۴-۱ b.

۴-۲ c. پرواز یک عمل یا رفتار در هواپیما است و نشاندهنده صفت نمی‌باشد.

۴-۳ به این معنی است که **count** از نوع **Integer** بوده و مقدار اولیه آن 500 می‌باشد. این صفت تعداد اسکناس‌های که هر روز در **CashDispenser** قرار داده می‌شود را کنترل می‌کند.

خودآزمایی

۴-۱ جاهای خالی را در عبارات زیر با کلمات مناسب پر کنید.

a) تمام برنامه‌ها در سه نوع عبارت کنترلی: .....، ..... و ..... نوشته شوند.

b) عبارت انتخاب ..... در صورت برقرار بودن شرط عملی را به اجرا درآورده و در صورت برقرار نبودن عمل دیگری را به اجرا در می‌آورد.

c) تکرار مجموعه‌ای از دستورالعمل‌ها به دفعات مشخص، تکرار ..... نامیده می‌شود.

d) زمانیکه از همان ابتدا تعداد تکرار عبارت مشخص نباشد، یک مقدار ..... می‌تواند به کار گرفته شده و به تکرار خاتمه دهد.

e) مشخص کردن ترتیب اجرای عبارت در یک برنامه کامپیوتری، ..... برنامه نامیده می‌شود.

f) ..... یک زبان مصنوعی و فرمال است که به برنامه‌نویسان در ایجاد الگوریتم‌ها کمک می‌کند.

g) ..... توسط زبان C++ به منظور پیاده سازی ویژگی‌های متفاوتی، نظیر عبارتهای کنترل، رزرو شده‌اند.

h) عبارت انتخابی ..... عبارت چند انتخابی نامیده می‌شود چرا که از میان موارد متفاوت انتخاب خود را انجام می‌دهد.

۴-۲ چهار عبارت متفاوت C++ بنویسید که 1 را به متغیر **x** از نوع صحیح اضافه کند.

۴-۳ عبارت یا مجموعه‌ای از عبارات را برای انجام موارد خواسته شده زیر بنویسید:

a) تخصیص مقدار **x** و **y** به **z** و سپس افزایش **x** به میزان یک واحد پس از انجام محاسبات. فقط با استفاده از یک عبارت.

b) بزرگتر بودن مقدار متغیر **count** را از 10 بررسی کنید. اگر چنین باشد، عبارت "Count is greater than 10" چاپ گردد.

c) متغیر **x** را از 1 کم کنید. سپس آنرا از متغیر **total** تفریق نمائید. فقط از یک عبارت استفاده کنید.

۴-۴ یک عبارت C++ برای انجام موارد خواسته شده زیر بنویسید:

a) اعلان متغیرهای **sum** و **x** از نوع **int**.

b) تخصیص 1 به متغیر **x**.

c) تخصیص 0 به متغیر **sum**.

d) محاسبه مجموع متغیرهای **x** و **sum** و تخصیص نتیجه به متغیر **sum**.

e) چاپ عبارت "**The sum is**:" که بدنبال آن مقدار متغیر **sum** آمده باشد.

۴-۵ با ترکیب عبارات نوشته شده در تمرین ۴-۴ آنرا به فرم برنامه‌ای در آورید که مجموع اعداد از 10 تا 1 را محاسبه و چاپ کند. از یک عبارت **while** برای ایجاد حلقه استفاده کنید. زمانیکه مقدار متغیر **x** به 11 رسید، حلقه پایان پذیرد.

۴-۶ مقدار هر متغیر را پس از هر محاسبه تعیین کنید. فرض کنید تمام متغیرها در ابتدای کار مقدار 5 دارند.

a) `product *= x++;`

b) `quotient /= ++x;`

۴-۷ عباراتی در C++ بنویسید که موارد خواسته شده زیر را انجام دهند.

a) وارد کردن متغیر صحیح **x** با **cin** و <<.

b) وارد کردن متغیر صحیح **y** با **cin** و <<.

c) مقداردهی متغیر صحیح **i** با **1**.

d) مقداردهی متغیر صحیح **power** با **1**.

e) ضرب متغیر **power** در **x** و تخصیص نتیجه به **power**.

f) پس افزایشی کردن متغیر **i** به میزان **1** واحد.

g) تعیین اینکه آیا **i** کوچکتر یا مساوی **y** است یا خیر.

h) چاپ متغیر صحیح **power** با **cin** و <<.

۴-۸ برنامه‌ای در C++ بنویسید که از عبارت تمرین ۴-۷ استفاده کرده و **x** را به توان **y** برساند. این برنامه باید از حلقه تکرار **while** استفاده کند.

۴-۹ خطا یا خطاهای موجود در عبارات زیر را تشخیص داده و اصلاح کنید.

(a

```
while (c <= 5)
      {
```

```
product *= c;
c++;
```

(b

```
cin<< value
```

(c

```
if (gender == 1)
    cout<<"Woman"<<endl;
else;
    cout<<"Man"<<endl;
```

۴-۱۰ در حلقه تکرار **while** عبارت زیر چه اشتباهی وجود دارد؟

```
while( z>= 0)
     sum +=z;
```

پاسخ خودآزمایی

۴-۱ a) توالی، انتخاب، تکرار. b) **if..else**. c) مشخص شده یا تعریف شده. d) مراقبتی، سیگنال یا پرچم. e) کنترل. f) شبه‌کد. g) کلمات کلیدی. h) **switch**.

۴-۲

```
x = x + 1;
x += 1;
++x;
x++;
```

۴-۳

(a
```
z = x++ + y;
```

(b
```
if (count > 10)
        cout<<"Count is greater than 10"<<endl;
```

(c
```
total -= --x;
```

۴-۴

(a
```
int sum;int x;
```

(b
```
x = 1;
```

(c
```
sum = 0;
```

(d
```
sum += x یا sum = sum + x;
```

(e
```
cout<<"The sum is:" <<Sum << endl;
```

۴-۵

```
// Exercise 4.5 Solution: ex04_05.cpp
// Calculate the sum of the integers from 1 to 10.
#include <iostream>
using std::cout;
using std::endl;
```

```
int main()
{
   int sum; // stores sum of integers 1 to 10
   int x; // counter

   x = 1; // count from 1
   sum = 0; // initialize sum

   while ( x <= 10 ) // loop 10 times
   {
      sum += x; // add x to sum
      x++; // increment x
   } // end while

   cout << "The sum is: " << sum << endl;
   return 0; // indicate successful termination
} // end main
```

۴-۶

a) `product = 25, x = 6;`

b) `quotient = 0, x = 6;`

```
// Exercise 4.6 Solution: ex04_06.cpp
// Calculate the value of product and quotient.
#include <iostream>
using std::cout;
using std::endl;

int main()
{
   int x = 5;
   int product = 5;
   int quotient = 5;

   // part a
   product *= x++; // part a statement
   cout << "Value of product after calculation: " << product << endl;
   cout << "Value of x after calculation: " << x << endl << endl;

   // part b
   x = 5; // reset value of x
   quotient /= ++x; // part b statement
   cout << "Value of quotient after calculation: " << quotient << endl;
   cout << "Value of x after calculation: " << x << endl << endl;
   return 0; // indicate successful termination
} // end main
```

۴-۷

a) cin >>x;
b) cin >>y;
c) i = 1;
d) power = 1;
e) power *=x ; or power = power * x ;
f) i++;
g) if (i<=y)
h) count << power << endl;

۴-۸

```
// Exercise 4.8 Solution: ex04_08.cpp
// Raise x to the y power.
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

int main()
{
   int x; // base
   int y; // exponent
   int i; // counts from 1 to y
   int power; // used to calculate x raised to power y

   i = 1; // initialize i to begin counting from 1
   power = 1; // initialize power

   cout << "Enter base as an integer: ";  // prompt for base
   cin >> x; // input base

   cout << "Enter exponent as an integer: "; // prompt for exponent
   cin >> y; // input exponent

   // count from 1 to y and multiply power by x each time
   while ( i <= y )
   {
      power *= x;
      i++;
   } // end while

   cout << power << endl; // display result
   return 0; // indicate successful termination
} // end main
```

٤-٩

a) خطا: فاقد براکت راست (}) در بدنه **while** است.

اصلاح : افزودن براکت راست پس از عبارت **;C++**

b) خطا: استفاده از عملگر درج بجای عملگر استخراج.

اصلاح: تغییر >> به <<.

c) خطا: سیمکولن پس از **else** یک خطای منطقی است. عبارت خروجی دوم همیشه اجرا خواهد شد.

اصلاح: حذف سیمکولن پس از **else**.

۴-۱۰ مقدار متغیر **z** هرگز در حلقه **while** تغییر نمی‌کند. از اینرو اگر شرط تکرار حلقه (**z >= 0**) در بدو امر برقرار باشد (**true**), یک حلقه بی‌نهایت بوجود خواهد آمد. برای اجتناب ازحلقه بی‌نهایت, بایستی **z** کاهش یابد تا سرانجام از صفر کمتر شود.

تمرینات

۴-۱۱ خطا یا خطاهای موجود در عبارت زیر را تشخیص داده و اصلاح کنید:

```
a) if (age >= 65);
       cout <<  "Age is greater than or equal to 65" << endl;
   else
       cout << "Age is less than to 65 << endl;"

b) if ( age >= 65);
      cout << "Age is greater than or equal to 65" << endl;
   else;
      cout << "Age is less than to 65" << endl";

c) int x = 1, total;

   while ( x <= 10)
  {
    total +=x;
    x++;
  }

d) while ( x <= 100 )
     total += x;
     x++;

e) while ( y > 0 )
   {
      cout << y << endl;
      y++;
   }
```

۴-۱۲ برنامه زیر چه عبارتی را چاپ می‌کند؟

```
// Exercise 4.12: ex04_12.cpp
// What does this program print?
#include <iostream>
using std::cout;
using std::endl;

int main()
{
   int y; // declare y
   int x = 1; // initialize x
   int total = 0; // initialize total

   while ( x <= 10 ) // loop 10 times
   {
      y = x * x; // perform calculation
      cout << y << endl; // output result
      total += y; // add y to total
      x++; // increment counter x
   } // end while

   cout << "Total is " << total << endl; // display result
   return 0; // indicate successful termination
} // end main
```

برای تمرینات ۴-۱۳ تا ۴-۱۶ هر یک از مراحل زیر را انجام دهید:

a) خواندن صورت مسئله.

b) فرموله کردن الگوریتم با شبه کد و مرحله اصلاح گام به گام از بالا به پایین.

c) نوشتن برنامه به زبان C++.

d) تست خطا یابی و اجرای برنامه.

۴-۱۳ راننده‌گان علاقه‌مند به دانستن مسافت طی شده توسط اتومبیل خود هستند. راننده‌ای مبادرت به ثبت تعداد دفعات سوخت‌گیری, میزان سوخت و مسافت پیموده شده می‌کند. برنامه‌ای در C++ بنویسد که با استفاده از یک حلقه while مبادرت به دریافت مسافت طی شده و تعداد گالون‌های زده شده در هر بار سوخت گیری کند. برنامه باید مسافت طی شده (مایل طی شده) بر حسب هر گالون را محاسبه کرده و بنمایش در آورد. خروجی برنامه می‌تواند شبیه عبارات زیر باشد.

```
Enter the miles used(-1 to quit):287
Enter gallons:13
MPG this tankful:22.076923
Total MPG:2.0769923

Enter the miles used(-1 to quit):200
Enter gallons:10
MPG this tankful:20.000000
Total MPG:21.173913

Enter the miles used(-1 to quit):-1
(MPG: Miles Per Gallon)
```

۴-۱۴ برنامه‌ای بنویسید که اگر مشتری یک فروشگاه بیش از موجودی خود در کارت اعتباری سفارش دهد، مشخص گردد. برای هر مشتری اطلاعات زیر موجود هستند:

۱- شماره حساب

۲- میزان موجودی در ابتدای ماه

۳- مجموع سفارشات از طرف مشتری در این ماه

۴- مجموع اعتبارات بکار برده شده توسط مشتری در این ماه.

۵- حد اعتبار

برنامه باید هر کدامیک از موارد فوق را دریافت و اعتبار جدید مشتری را محاسبه کند (= میزان موجودی + سفارشات - اعتبار) و مشخص نماید که آیا سفارش مشتری بیش از اعتبارش است یا خیر. در صورت سفارش بیش از حد پیغام "**Credit limit exceeded**" چاپ شود. خروجی برنامه می‌تواند همانند عبارات زیر باشد.

```
Enter account number(-1 to end):100
Enter beginning balance:5394.78
Enter total credits:500.00
Enter credit limit:5500.00
New balance is 5894.78
Account:100
Credit limit:5500.00
Balance:5894.78
```

```
Credit Limit Exceeded

Enter account number(-1 to end):200
Enter beginning balance:1000.00
Enter total charges:123.45
Enter credit limit:1500.00
New balance is 802.45

Enter account number(-1 to end):300
Enter beginning balance:500.00
Enter total charges:274.73
Enter total credits:100.00
Enter credit limit:800.00
New balance is 674.73
```

۴-۱۵ یک شرکت بزرگ شیمیایی براساس کمیسیون، حقوق فروشندگان خود را پرداخت می‌کند. هر فروشنده‌ای، در هفته 200 دلار به همراه 9 درصد از فروش هفته دریافت می‌کند. برای مثال، فروشنده‌ای که 5000 دلار در هفته فروش داشته، 200 دلار به همراه 9 در صد از 5000 دلار یا مجموع 650 دلار دریافت می‌کند. برنامه‌ای بنویسید که از یک عبارت **while** برای دریافت فروش هفتگی هر فروشنده استفاده کرده و دریافتی وی را به نمایش در آورد. در هر بار، حقوق یک فروشنده را محاسبه نماید.

```
Enter sales in dollars(-1 to end):5000.00
Salary is:$650.00
Enter sales in dollars(-1 to end):6000.00
Salary is:$740.00
Enter sales in dollars(-1 to end):-1
```

۴-۱۶ برنامه‌ای در C++ بنوسید که با استفاده از یک عبارت while حقوق دریافتی چند کارمند را محاسبه کند. شرکت بر اساس 40 ساعت کار در هفته حقوق مستقیم پرداخت می‌کند و اگر بیش از 40 ساعت کار صورت گرفته باشد، نصف آن زمان به حقوق 40 ساعتی افزوده می‌شود. برنامه اطلاعات هر کارمند را که شامل ساعت کار کرده در هفته بوده به همراه نرخ دستمزد ساعتی را دریافت کرده و حقوق وی را محاسبه می‌کند. خروجی برنامه می‌تواند همانند عبارات زیر باشد.

```
Enter hours worked(-1 to end):39
Enter hourly rate of the worker($00.00):10.00
Salary is $390.00

Enter hours worked(-1 to end):40
Enter hourly rate of the worker($00.00):10.00
Salary is $400.00

Enter hours worked(-1 to end):41
Enter hourly rate of the worker($00.00):10.00
Salary is $415.00

Enter hours worked(-1 to end):-1
```

۴-۱۷ فرآیند یافتن بزرگترین عدد از جمله برنامه‌های پرکاربرد است. برای مثال، برنامه‌ای می‌تواند بهترین فروشنده را بر اساس میزان فروش تعیین کند. کسی که بیشترین فروش را داشته است، به عنوان برنده انتخاب می‌شود.

شبه‌کدی نوشته، سپس برنامه C++ آنرا ایجاد کنید که با استفاده از عبارت **while** مبادرت به تعیین و چاپ بزرگترین عدد از بین **10** عدد وارد شده توسط کاربر کند. برنامه باید از سه متغیر به شرح زیر استفاده نماید:

counter: یک شمارنده برای شمارش تا **10**

number: عدد جاری وارد شده به برنامه.

largest: بزرگترین عدد دریافت شده تا بدین جا.

۴-۱۸ برنامه‌ای بنویسید که با استفاده از عبارت **while** اقدام به چاپ مقادیر جدول زیر کند:

N	10*N	100*N	1000*N
1	10	100	1000
2	20	200	2000
3	20	300	3000
4	40	400	4000
5	50	500	5000

۴-۱۹ با استفاده از روش بکار رفته در تمرین ۴-۱۷، دو عدد بزرگ را در میان 10 عدد پیدا کنید (نکته: باید هر عدد را یک بار وارد کنید.)

۴-۲۰ برنامه بررسی نتایج آزمون در شکل‌های ۴-۱۶ الی ۴-۱۸ فرض می‌کند که هر مقدار ورودی که توسط کاربر وارد می‌شود اگر 1 نباشد پس 2 است. برنامه را برای اعتبارسنجی ورودی‌ها اصلاح کنید. اگر مقدار وارد شده 1 یا 2 نباشد، حلقه تا دریافت مقدار صحیح تکرار شود.

۴-۲۱ برنامه زیر چه عبارتی را چاپ می‌کند؟

```
// Exercise 4.21: ex04_21.cpp
// What does this program print?
#include <iostream>
using std::cout;
using std::endl;

int main()
{
   int count = 1; // initialize count

   while ( count <= 10 ) // loop 10 times
   {
      // output line of text
      cout << ( count % 2 ? "****" : "++++++++" ) << endl;
      count++; // increment count
   } // end while

   return 0; // indicate successful termination
} // end main
```

۴-۲۲ برنامه زیر چه عبارتی را چاپ می‌کند؟

```
// Exercise 4.22: ex04_22.cpp
// What does this program print?
#include <iostream>
using std::cout;
using std::endl;

int main()
{
   int row = 10; // initialize row
   int column; // declare column

   while ( row >= 1 ) // loop until row < 1
   {
      column = 1; // set column to 1 as iteration begins

      while ( column <= 10 ) // loop 10 times
      {
         cout << ( row % 2 ? "<" : ">" ); // output
         column++; // increment column
      } // end inner while

      row--; // decrement row
      cout << endl; // begin new output line
   } // end outer while

   return 0; // indicate successful termination
} // end main
```

۴-۲۳ مشکل Danglin-Else. خروجی هر یک از عبارات زیر را با فرض زمانیکه x برابر 9، y برابر 11 و زمانیکه x برابر 11 و y برابر 9 است را تعیین کنید. توجه کنید که کامپایلر دندانه‌گذاری موجود در یک برنامه C++ را نادیده می‌گیرد. کامپایلر C++ همیشه یک else را با if قبلی در نظر می‌گیرد مگر اینکه با قرار دادن براکت‌ها {} غیر این را حکم کنید. در نگاه اول، برنامه‌نویس نمی‌تواند مطمئن باشد که کدام if و else با هم هستند، از اینرو این مشکل Dangling-else شناخته می‌شود. برای اینکه حل مسئله کمی مشکل‌ت شود، دندانه‌گذاری را در این عبارات اعمال نکرده‌ایم.

a)

```
if (x<10)
if (y>10)
cout << "*****"<<endl;
else
cout <<"#####"<<endl;
cout <<"$$$$$"<<endl;
```

b)

```
if (x<10)
{
if (y>10)
cout <<"*****"<<endl;
}
else
{
```

```
cout <<"#####"<<endl;
cout <<"$$$$$"<<endl;
}
```

۴-۲۴ (مشکل Dangling-Else) کد زیر را برای تولید خروجی به نمایش در آمده اصلاح کنید. از دندانه‌گذاری مناسب استفاده کنید. هیچ تغییری بجز اعمال براکت نباید در کد بوجود آورید.

```
if (y == 8)
if (x == 5)
cout << "@@@@@" << endl;
else
cout << "#####" << endl;
cout << "$$$$$" << endl;
cout << "&&&&&" << endl;
```

a) با فرض x=5 و y=8 ،خروجی زیر تولید شود:

```
@@@@@
$$$$$
&&&&&
```

b) با فرض x=5 و y=8، خروجی زیر تولید شود:

```
@@@@@
```

c) با فرض x=5 و y=8، خروجی زیر تولید شود:

```
@@@@@
&&&&&
```

d) با فرض x=5 و y=7، خروجی زیر تولید شود:

```
#####
$$$$$
&&&&&
```

۴-۲۵ برنامه‌ای بنویسید که سایز یک ضلع چهارگوش را دریافت و یک چهارگوش توخالی براساس آن سایز از ستاره‌ها (*) و فاصله‌ها چاپ کند. برنامه باید برای ترسیم چهارگوش‌های با سایز 1 تا 20 عمل کند. برای مثال، اگر سایز 5 وارد برنامه شود، بایستی خروجی زیر چاپ شود.

```
*****
*   *
*   *
*   *
*****
```

۴-۲۶ پالندروم، عدد یا عبارتی است که خواندن آن از هر دو جهت یکسان است. برای مثال، اعداد پنج رقمی و از نوع صحیح زیر همگی پالندروم هستند: 12321، 55555، 45554، 11611. برنامه‌ای بنویسید که پنج رقم از نوع صحیح دریافت کرده و تعیین کند که آیا پالندروم است یا خیر.

۴-۲۷ یک عدد صحیح فقط حاوی صفرها و یک‌ها (یعنی باینری) دریافت کرده و معادل دیسمال آنرا چاپ کنید. از عملگر باقیمانده و تقسیم برای انتخاب ارقام باینری از سمت راست به چپ استفاده کنید (در هر بار یک رقم). با توجه به اینکه در سیستم عددی دیسمال، سمت راست‌ترین رقم دارای ارزش مکانی 1، رقم بعدی دارای ارزش مکانی 10، سپس 100، سپس 1000 و الی آخر است. در سیستم عددی باینری، سمت راست‌ترین رقم دارای ارزش مکانی 1، رقم بعدی دارای ارزش مکانی 2، سپس 4، سپس 8 و الی آخر است. از اینرو عدد دیسمال 234 می‌تواند

بصورت 1*4 + 3*10 + 2*100 تفسیر شود. معادل دیسمال عدد باینری 1101 است که بصورت 1*4 + 2*0 + 1*1 + 1*8 یا 8 + 4 + 0 + 1 یا 13 است.

۴-۲۸ برنامه بنویسید که الگوی زیر را به نمایش در آورد. برنامه باید از سه عبارت خروجی استفاده کرده باشد، یکی از عبارات می‌تواند بفرم زیر باشد:

```
   cout << "*" ;
   cout << '  ' ;
   cout << endl ;
********
 ********
********
 ********
********
 ********
********
 ********
```

۴-۲۹ برنامه‌ای بنویسید که توالی‌هایی از 2 را بصورت 2,4,8,16,32,64 الی آخر تولید کرده و چاپ نماید. حلقه **while** نباید خاتمه یابد (یعنی یک حلقه بی‌نهایت ایجاد کنید). برای انجام اینکار، کافیست در شرط عبارت **while** کلمه کلیدی **true** را قرار دهید. با انجام اینکار چه اتفاقی رخ خواهد داد؟

۴-۳۰ برنامه‌ای بنویسید که شعاع یک دایره را دریافت (از نوع **double**) و قطر، مساحت و محیط آن را محاسبه کنید. از مقدار 3.14159 برای π استفاده کنید.

۴-۳۱ در عبارت زیر چه اشتباهی وجود دارد؟ عبارت زیر را به نحوی اصلاح کنید که خواسته برنامه‌نویس را برآورده سازد.

```
cout << ++(x+y);
```

۴-۳۲ برنامه‌ای بنویسید که مقدار double غیرصفرخوانده و تعیین کند که این مقادیر می‌توانند نشاندهنده اضلاع یک مثلث باشند یا خیر.

۴-۳۳ برنامه‌ای بنویسید که مقدار صحیح غیرصفر خوانده و تعیین کند که این مقادیر می‌توانند نشاندهنده یک مثل راست گوشه باشند یا خیر.

۴-۳۴ شرکتی می‌خواهد داده‌های خود را از طریق خط تلفن منتقل نماید. تمام داده‌های انتقالی از چهار رقم صحیح تشکیل شده‌اند. برنامه‌ای بنویسید که داده‌های انتقالی این شرکت را بصورت کد درآورد. ابتدا برنامه یک عدد چهار رقمی صحیح را خوانده و سپس بطریق زیر آنرا کدگذاری نماید: هر رقم را با باقیمانده تقسیم بر 10 جایگزین سازد. سپس مکان رقم اول را با رقم سوم و رقم چهارم را با رقم دوم عوض کند. در پایان عدد کدگذاری شده را چاپ کند. در ادامه برنامه‌ای بنویسید که عدد کدگذاری شده را دریافت و آنرا کدگشایی نماید.

۴-۳۵ فاکتوریل n عدد صحیح غیرمنفی بصورت !n نوشته می‌شود و بصورت زیر تعریف می‌گردد:

`n! = n.(n - 1).(n - 2).....1` (n = مقادیر بزرگتر یا مساوی 1)

و

n!= 1 (برای n = 0)

برای مثال 5.4.3.2.1 = !5 است که حاصل آن 120 است.

a) برنامه‌ای بنویسید که یک مقدار صحیح غیرمنفی را دریافت (از طریق کادر تبادلی) و فاکتوریل آنرا محاسبه و چاپ کند.

b) برنامه‌ای بنویسید که مقدار ثابت ریاضی e را با استفاده از فرمول زیر تخمین بزند:

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots$$

c) برنامه‌ای بنویسید که مقدار e^x را با استفاده از فرمول زیر محاسبه کند:

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$$

فصل پنجم

عبارات کنترلی: بخش ۲

اهداف

- آشنایی با عبارات تکرار for، do...while و اجرای عبارات تکرار شونده.
- اصول شمارنده کنترل تکرار.
- استفاده از عبارت چند انتخابی switch.
- استفاده از عبارات کنترل برنامه break و continue.
- استفاده از عملگرهای منطقی.
- اجتناب از پی‌آمد اشتباه گرفتن عملگر تخصیص با تساوی.

رئوس مطالب

۵-۱ مقدمه

فصل چهارم را با معرفی انواع بلوک‌های سازنده که در حل مسئله نقش دارند، آغاز کردیم. با استفاده از این بلوک‌های سازنده تکنیک‌های ایجاد برنامه را بهبود بخشیدیم. در این فصل، مبحث تئوری و قواعد علمی برنامه‌نویسی ساخت‌یافته را با معرفی مابقی عبارات کنترلی C++ ادامه می‌دهیم. با عبارات کنترلی که در این فصل مطرح می‌کنیم و عبارات کنترلی معرفی شده در فصل قبلی قادر به ایجاد و کنترل شی‌ها خواهیم بود. همچنین به مبحث برنامه‌نویسی شی‌گرا که آن را از فصل اول آغاز کرده‌ایم ادامه می‌دهیم.

در این فصل به توصیف عبارات **do...while**, **for** و **switch** می‌پردازیم. در کنار مثال‌های کوچکی که در آنها از **while** و **for** استفاده شده، به بررسی اصول و نیازهای شمارنده-کنترل‌تکرار خواهیم پرداخت. بخشی از این فصل را اختصاص به گسترش کلاس **GradeBook** عرضه شده در فصل‌های سوم و چهارم داده‌ایم. در واقع، نسخه‌ای از کلاس **GradeBook** را ایجاد می‌کنیم که از عبارت **switch** برای شمارش تعداد نمرات A، B، C، D و F وارد شده از سوی کاربر استفاده می‌کند. به معرفی عبارات **break** و**continue** خواهیم پرداخت که کنترل‌کننده برنامه هستند. در مورد عملگرهای منطقی، که به برنامه‌نویسان اجازه می‌دهند تا از شرط‌های پیچیده و قدرتمندتر در عبارات کنترلی استفاده کنند، صحبت خواهیم کرد.

همچنین در ارتباط با خطای رایجی که در ارتباط با عدم درک صحیح تفاوت مابین عملگر برابری (==) و تخصیص (=) رخ می‌دهد توضیحاتی ارائه می‌کنیم. در پایان، بطور خلاصه شده به توصیف عبارات کنترلی C++ و تکنیک‌های حل مسئله مطرح شده در این فصل و فصل چهارم می‌پردازیم.

۵-۲ نکاتی در مورد شمارنده-کنترل تکرار

در فصل قبل، با مفهوم روش شمارنده-کنترل تکرار آشنا شدید. در این بخش با استفاده از عبارت تکرار while، اقدام به فرموله کردن عناصر مورد نیاز در روش شمارنده-کنترل تکرار می‌کنیم:

۱- نام ***متغیر کنترل*** (یا شمارنده حلقه) که برای تعیین تکرار حلقه بکار گرفته می‌شود.

۲- ***مقدار اولیه*** متغیر کنترل.

۳- ***شرط تکرار حلقه*** برای تست مقدار نهایی متغیر کنترل (آیا حلقه ادامه یابد یا خیر).

۴- ***افزایش*** (یا ***کاهش***) متغیر کنترل در هر بار تکرار حلقه.

در برنامه شکل ۱-۵ از چهار عنصر شمارنده-کنترل تکرار برای نمایش ارقام 1-10 استفاده شده است. در خط 9، نام *متغیر کنترلی* **(counter)** از نوع صحیح اعلان شده و فضای در حافظه برای آن رزرو می‌شود و با *مقدار اولیه* 1 تنظیم شده است. این اعلان یک مقداردهی اولیه است. بخش مقداردهی این عبارت یک جزء اجرائی است و از اینرو، خود عبارت هم اجرائی می‌باشد.

اعلان و مقداردهی اولیه **counter** در خط 9 را می‌توان توسط عبارات زیر هم انجام داد:

```
int counter; // declare control variable
counter = 1; // initialize control variable to 1
```

ما از هر دو روش استفاده می‌کنیم.

به عبارت **while** (خطوط 15-11) توجه کنید. خط 14 مقدار جاری **counter** را به میزان 1 واحد در هر بار تکرار حلقه افزایش می‌دهد. شرط تکرار حلقه (خط 11) در عبارت **while** تست می‌کند که آیا مقدار متغیر کنترل کمتر یا برابر 10 است یا خیر، به این معنی که 10، *مقدار نهایی* برای برقرار بودن شرط است. بدنه عبارت **while** تا زمانیکه متغیر کنترل به 5 برسد اجرا می‌شود. زمانیکه مقدار متغیر کنترل بیش از 10 شود (هنگامی که **counter** به مقدار 11 برسد)، حلقه پایان می‌یابد.

```
1  // Fig. 5.1: fig05_01.cpp
2  // Counter-controlled repetition.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     int counter = 1; // declare and initialize control variable
10
```

```
11    while ( counter <= 10 ) // loop-continuation condition
12    {
13       cout << counter << " ";
14       counter++; // increment control variable by 1
15    } // end while
16
17    cout << endl; // output a newline
18    return 0; // successful termination
19 } // end main
```

```
 1 2 3 4 5
```

شکل ۱-۵ | شمارنده-کنترل تکرار.

برنامه‌نویسی ایده‌ال

قبل و بعد از هر عبارت یک خط خالی قرار دهید تا قسمت‌های متفاوت برنامه از یکدیگر تمیز داده شوند.

برنامه‌نویسی ایده‌ال

قراردادن فضاهای خالی در ابتدا و انتهای عبارتهای کنترلی و دندانه‌دار کردن این عبارتها به برنامه یک ظاهر دوبعدی می‌دهد و باعث افزایش خوانایی برنامه می‌شود.

با مقداردهی **counter** با 0 و جایگزین کردن عبارت **while** با

```
while( ++counter <= 10 ) //loop-continuation condition
   cout << counter << " ";
```

می‌توان برنامه شکل ۱-۵ را مختصرتر کرد. با این کد از یک عبارت صرفه‌جویی شده است، چرا که عملیات افزایش بصورت مستقیم در شرط **while** قبل از بررسی شرط صورت می‌گیرد. همچنین این کد نیاز به استفاده از براکت‌ها در اطراف بدنه **while** را از بین برده است، چرا که **while** فقط حاوی یک عبارت است. گاهی اوقات فشرده‌سازی کد به این روش می‌تواند خوانایی، نگهداری، اصلاح و خطایابی برنامه را مشکل کند.

برنامه‌نویسی ایده‌ال

با افزایش تعداد سطح‌های تودرتو، درک عملکرد برنامه مشکل‌تر می‌شود. بعنوان یک قانون، سعی کنید از سه سطح تودرتو فراتر نروید.

خطای برنامه‌نویسی

مقادیر اعشاری تقریبی هستند، از اینرو کنترل شمارش حلقه‌ها با متغیرهای اعشاری می‌تواند در شمارش مقادیر، غیردقیق بوده و شرط خاتمه را بدقت انجام ندهند.

اجتناب از خطا

شمارش حلقه‌ها را با مقادیر صحیح کنترل کنید.

۵-۳ عبارت تکرار for

در بخش ۲-۵ به بررسی اصول شمارنده-کنترل تکرار پرداخته شد. می‌توان از عبارت**while** در پیاده‌سازی هر حلقه کنترل شده با شمارنده استفاده کرد. زبان C++ دارای عبارت تکرار **for** است که از تمام جزئیات

شمارنده-کنترل‌تکرار استفاده می‌کند. برای نشان دادن قدرت **for** برنامه ۱-۵ را مجدداً بازنویسی می‌کنیم. نتیجه اینکار در برنامه شکل ۲-۵ دیده می‌شود.

```
1  // Fig. 5.2: fig05_02.cpp
2  // Counter-controlled repetition with the for statement.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     // for statement header includes initialization,
10    // loop-continuation condition and increment.
11    for ( int counter = 1; counter <= 10; counter++ )
12       cout << counter << " ";
13
14    cout << endl; // output a newline
15    return 0; // indicate successful termination
16 } // end main
```

```
 1 2 3 4 5 6 7 8 9 10
```

شکل ۲-۵ | شمارنده-کنترل‌تکرار با عبارت for.

هنگامی که عبارت **for** شروع بکار می‌کند (خطوط 12-11)، متغیر کنترل **counter** با 1 مقداردهی می‌شود، از اینرو تا بدین جا دو عنصر اولیه شمارنده-کنترل تکرار یعنی *نام متغیر* و *مقدار اولیه* تعیین شده‌اند. سپس، شرط ادامه حلقه **counter <= 10** بررسی می‌شود. بدلیل اینکه مقدار اولیه متغیر **counter**، برابر 1 است، شرط برقرار بوده، و مقدار1 با اجرای عبارت خط 12 چاپ می‌شود. سپس مقدار متغیر **counter** توسط عبارت **counter++** افزایش یافته و مجددا حلقه با تست شرط تکرار آغاز می‌شود. در این لحظه، متغیر کنترل حاوی مقدار 2 است. این مقدار متجاوز از مقدار پایانی نیست و از اینرو برنامه عبارت موجود در بدنه را به اجرا در می‌آورد. این فرآیند تا زمانیکه **counter** به مقدار 10 برسد و آنرا چاپ کند و متغیر کنترل **counter** به 11 افزایش یابد، ادامه پیدا می‌کند و موجب می‌شود تا تست شرط حلقه برقرار نشده و تکرار خاتمه یابد. برنامه با اجرای اولین عبارت پس از عبارت **for** ادامه می‌یابد (در این مثال، برنامه به عبارت خروجی در خط 14 می‌رسد و آنرا اجرا می‌کند).

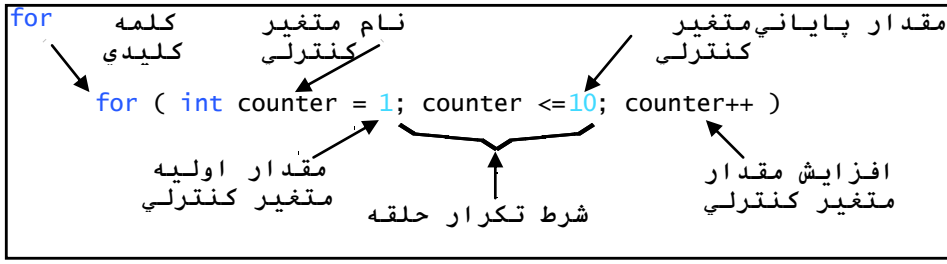
کامپونت‌های تشکیل دهنده سرآیند for

در شکل۳-۵ نگاهی دقیق‌تر بر عبارت **for** برنامه ۲-۵ داشته‌ایم (خط 11). گاهاً به خط اول عبارت **for** *سرآیند for* می‌گویند. این سرآیند حاوی ایتم‌های مورد نیاز در ایجاد شمارنده-کنترل‌تکرار به همراه یک متغیر کنترلی است.

دقت کنید که در برنامه۲-۵ از شرط تکرار حلقه **counter <= 10** استفاده شده است. اگر برنامه‌نویس اشتباهاً بنویسد **counter < 10**، حلقه فقط 9 بار اجرا خواهد شد. این خطا معروف به خطای *off-by-one* است.

خطای برنامه‌نویسی

استفاده از عملگر رابطه‌ای اشتباه یا استفاده از یک مقدار نهایی اشتباه در شرط شمارنده یک حلقه ***while*** *یا* ***for*** *می‌تواند خطای off-by-one بدنبال داشته باشد.*



شکل ۳-۵ | کامپونت‌های سرآیند for.

برنامه‌نویسی ایده‌ال

استفاده از مقدار نهایی در شرط یک عبارت ***while*** *یا* ***for*** *و استفاده از عملگر رابطه‌ای <= می‌تواند جلوی خطاهای off-by-one را بگیرید. برای مثال، در حلقه‌ای که برای چاپ مقادیر 1 تا 10 کاربرد دارد، شرط تکرار حلقه بایستی* ***counter <= 10*** *بجای* ***counter < 10*** *یا* ***counter < 11*** *باشد. برخی از برنامه‌نویسان ترجیح می‌دهند شمارش را از صفر آغاز کنند، در اینحالت برای شمارش 10 بار تکرار حلقه، بایستی متغیر* ***counter*** *با صفر مقداردهی شده و شرط تست حلقه به صورت* ***counter < 10*** *نوشته شود.*

فرمت کلی عبارت **for** بشکل زیر است

for (*مقداردهی اولیه; شرط تکرار حلقه; افزایش***)**
عبارت

در این عبارت، جمله مقداردهی اولیه نشاندهنده نام متغیر کنترلی حلقه بوده و مقدار اولیه را فراهم می‌آورد، شرط تکرار حلقه حاوی مقدار پایانی متغیر کنترلی بوده و در صورت برقرار بودن حلقه به اجرا در می‌آید، و جمله افزایش مبادرت به افزودن مقدار متغیر کنترلی می‌کند. می‌توان بجای عبارت **for** از معادل عبارت **while** و بصورت زیر استفاده کرد:

مقداردهی اولیه;
while (*شرط تکرار حلقه***) {**
عبارت
افزایش;
}

در اینجا فقط یک استثناء وجود دارد که در بخش ۷-۵ به بررسی آن خواهیم پرداخت.

اگر جمله مقداردهی اولیه در سرآیند عبارت **for** اعلان‌کننده متغیر کنترلی باشد (نوع متغیر کنترلی قبل از نام متغیر آمده باشد)، می‌توان از آن متغیر کنترلی فقط در بدنه همان **for** استفاده کرد. این متغیر کنترلی در خارج از عبارت **for** شناخته شده نخواهد بود. این محدودیت استفاده از نام متغیر کنترلی بعنوان *قلمرو* متغیر شناخته می‌شود. قلمرو یک متغیر تعیین‌کننده مکانی است که متغیر می‌تواند در برنامه بکار گرفته شود. در فصل ششم با جزئیات قلمرو آشنا خواهید شد.

خطای برنامه‌نویسی

*زمانیکه متغیر کنترلی در یک عبارت **for** در بخش مقداردهی اولیه سرآیند **for** اعلان شده باشد، استفاده از آن متغیر کنترلی پس از عبارت، یک خطای کامپایل بدنبال خواهد داشت.*

قابلیت حمل

*در ++C استاندارد، قلمرو متغیر کنترلی اعلان شده در بخش مقداردهی اولیه یک عبارت **for** با قلمرو مطرح شده در کامپایلرهای قدیمی ++C متفاوت است. در کامپایلرهای قدیمی، قلمرو متغیر کنترلی با پایان یافتن بلوک تعیین کننده عبارت **for** خاتمه نمی‌پذیرد. کد ++C ایجاد شده با کامپایلرهای قدیمی ++C می‌تواند به هنگام کامپایل با کامپایلرهای استاندارد با مشکل مواجه شود. اگر در حال کار با کامپایلرهای قدیمی هستید و می‌خواهید از عملکرد دقیق کدهای خود بر روی کامپایلرهای استاندارد مطمئن شوید، دو استراتژی برنامه‌نویسی تدافعی وجود دارد که می‌توانید از آنها استفاده کنید: اعلان متغیرهای کنترلی با اسامی مختلف در هر عبارت **for** یا اگر ترجیح می‌دهید از نام یکسانی برای متغیر کنترلی در چندین عبارت **for** استفاده کنید، اعلان متغیر کنترلی قبل از اولین عبارت **for** است.*

همانطوری که مشاهده خواهید کرد، جملات مقداردهی اولیه و افزایش را می‌توان با کاما از یکدیگر متمایز کرد. کامای (ویرگول) بکار رفته در این جملات، از نوع عملگرهای کاما هستند و تضمین می‌کنند که لیست جملات از سمت چپ به راست ارزیابی خواهند شد. عملگر کاما از تمام عملگرهای ++C از تقدم پایین‌تری برخوردار است. مقدار و نوع یک لیست جدا شده با کاما، معادل مقدار و نوع سمت راست‌ترین جمله در لیست خواهد بود. در اکثر مواقع از کاما در عبارات **for** استفاده می‌شود. اصلی‌ترین کاربرد کاما این است که برنامه‌نویس امکان استفاده از چندین جمله مقداردهی اولیه و یا چندین جمله افزایش‌دهنده را فراهم می‌آورد. برای مثال، امکان دارد چندین متغیر کنترلی در یک عبارت **for** وجود داشته باشند که بایستی مقداردهی اولیه شده و افزایش داده شوند.

مهندسی نرم‌افزار

با قرار دادن یک سیمکولن بلافاصله پس از سرآیند for، یک حلقه تاخیردهنده بوجود می‌آید. چنین حلقه‌ای با یک بدنه تهی فقط به تعداد دفعات مشخص شده تکرار شده و کاری بجز شمارش انجام نمی‌دهد. برای مثال، امکان دارد از یک حلقه تاخیری برای کاستن از سرعت قراردادن خروجی در صفحه نمایش استفاده کنید تا

کاربر بتواند براحتی آن را بخواند. اما باید مراقب باشید، چرا که چنین زمان تاخیری در میان انواع سیستم‌ها با پردازنده‌های مختلف متفاوت است.

سه عبارت در عبارت **for** حالت اختیاری دارند. اگر شرط تکرار حلقه فراموش شود، C++ فرض خواهد کرد که شرط تکرار حلقه برقرار است، و از اینرو یک حلقه بی‌نهایت بوجود خواهد آمد. می‌توان عبارت مقداردهی اولیه متغیر کنترلی را از عبارت **for** خارج کرد، اگر این متغیر در جای دیگری از برنامه و قبل از حلقه مقداردهی شده باشد. عبارت سوم مربوط به بخش افزایش است و در صورتیکه عملیات افزایش مقدار متغیر کنترلی توسط عبارتی در بدنه **for** صورت گیرد یا نیازی به افزایش وجود نداشته باشد، می‌توان آنرا از سرآیند حذف کرد. عبارت افزایشی در عبارت **for** همانند یک عبارت منفرد در انتهای بدنه **for** عمل می‌کند. از اینرو، عبارات

```
counter = counter + 1
counter += 1
++counter
counter++
```

همگی معادل بخش افزایشی در سرآیند عبارت **for** هستند. بسیاری از برنامه‌نویسان ترجیح می‌دهند تا از **counter++** استفاده کنند. به این دلیل که عملیات افزایش مقدار متغیر کنترلی را پس از اجرای بدنه حلقه به انجام می‌رساند، و از اینرو این رفتار در افزایش بسیار طبیعی بنظر می‌رسد.

خطای برنامه‌نویسی

*استفاده از کاما بجای سیمکولن در سرآیند عبارت **for** خطای نحوی خواهد بود.*

خطای برنامه‌نویسی

*قرار دادن یک سیمکولن بلافاصله در سمت راست پرانتز یک سرآیند **for** یک بدنه تهی برای **for** تولید خواهد کرد. اینکار می‌تواند خطای منطقی بدنبال داشته باشد.*

بخش‌های مقداردهی اولیه، شرط تکرار حلقه و افزایش در عبارت **for** می‌توانند حاوی عبارات محاسباتی باشند. برای مثال، فرض کنید که $x = 2$ و $y = 10$ باشد. اگر x و y در داخل بدنه حلقه تغییری نیابد، پس عبارت

```
for ( int j = x; j <= 4 * x * y; j += y / x)
```

معادل عبارت زیر خواهد بود

```
for ( int j = 2; j <= 80; j += 5)
```

مقدار بخش افزایش می‌تواند منفی باشد، در اینحالت گام پیمایش حلقه معکوس خواهد بود. اگر شرط تکرار حلقه در همان ابتدای کار برقرار نباشد، بدنه عبارت **for** اجرا نخواهد شد، و اجرا با عبارت پس از

عبارت **for** ادامه می‌یابد. مقدار متغیر کنترلی به دفعات در محاسبات درون بدنه عبارت **for** بکار گرفته می‌شود یا به نمایش در می‌آید، اما اینکار الزامی ندارد. در بسیاری از موارد از متغیر کنترلی فقط برای کنترل تکرار استفاده می‌شود.

اجتناب از خطا

*اگر چه مقدار، متغیر کنترلی می‌تواند در بدنه حلقه **for** تغییر یابد، اما از انجام چنین کاری اجتناب کنید چرا که می‌تواند برنامه را بسمت خطاهای ناخواسته‌ای هدایت کند.*

دیاگرام فعالیت UML عبارت for

دیاگرام فعالیت UML عبارت **for** شبیه به دیاگرام فعالیت **while** است (شکل ۶-۴). در شکل ۴-۵ دیاگرام فعالیت عبارت **for** بکار رفته در برنامه ۲-۵ آورده شده است. در این دیاگرام مشخص است که فرآیند مقداردهی اولیه فقط یکبار و قبل از ارزیابی تست شرط تکرار حلقه برای بار اول صورت می‌گیرد و اینکه عمل افزایش در هر بار و پس از اجرای عبارات بدنه انجام می‌شود. دقت کنید (در کنار وضعیت اولیه، خطوط انتقال، ادغام، وضعیت پایانی) دیاگرام فقط حاوی یک وضعیت عمل و یک تصمیم‌گیری است.

شکل ٤-٥ | دیاگرام فعالیت UML عبارت تکرار for در برنامه ۲-۵.

٤-٥ مثال‌های با استفاده از عبارت for

مثال‌های مطرح شده در این بخش، متدهای متنوعی از کاربرد متغیر کنترلی در یک عبارت **for** هستند. در هر مورد، سرآیند **for** نوشته شده است.

a) متغیر کنترلی که از 1 تا 100 با گام 1 افزایش می‌یابد.

```
for ( int i = 1; i <= 100; i++)
```

b) متغیر کنترلی که از 100 تا 1 با گام 1- افزایش می‌یابد (با کاهش 1).

```
for ( int i = 100; i >= 1; i--)
```

c) متغیر کنترلی که از 7 تا 77، با گام 7 افزایش می‌یابد.

```
for ( int i = 7; i <= 77; i += 7)
```

d) متغیر کنترلی که از 20 تا 2 با گام 2- افزایش می‌یابد.

```
for ( int i = 20; i >= 2; i -=2)
```

e) متغیر کنترلی، که توالی از مقادیر 20، 17، 14، 11، 8، 5 و 2 به خود می‌گیرد.

```
for ( var j = 2; j <= 20; j += 3)
```

f) متغیر کنترلی که توالی از مقادیر 0، 11، 22، 33، 44، 55، 66، 77، 88 و 99 به خود می‌گیرد.

```
for ( var j = 99; j >= 20; j -=11)
```

خطای برنامه‌نویسی

نتیجه عدم استفاده صحیح از عملگر رابطه‌ای در شرط تکرار حلقه که بصورت معکوس شمارش می‌کند (همانند استفاده اشتباه ***i <= 1*** *بجای* ***i >= 1*** *در شمارش معکوس حلقه به سمت 1) معمولاً یک خطای منطقی است که نتایج اشتباهی به هنگام اجرا برنامه تولید می‌کند.*

برنامه: مجموع اعداد زوج از 2 تا 20

دو مثال بعدی برنامه‌های ساده‌ای هستند که به توضیح عبارت **for** می‌پردازند. برنامه ۵-۵ از عبارت **for** برای بدست آوردن مجموع اعداد زوج 100 تا 2 استفاده می‌کند. در هر بار تکرار حلقه (خطوط 13-12) مقدار جاری متغیر کنترل **number** به متغیر **total** افزوده می‌شود.

```
1  // Fig. 5.5: fig05_05.cpp
2  // Summing integers with the for statement.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     int total = 0; // initialize total
10
11    // total even integers from 2 through 20
12    for ( int number = 2; number <= 20; number += 2 )
13       total += number;
14
15    cout << "Sum is " << total << endl; // display results
16    return 0; // successful termination
17 } // end main
```

```
Sum is 110
```

شکل ۵-۵ | عبارت for بکار رفته برای محاسبه مجموع اعداد زوج.

به بدنه عبارت **for** در برنامه ۵-۵ توجه نمائید. در واقع می‌توان این بدنه را با سمت راستترین بخش سرآیند **for** و با استفاده از *کاما* ادغام کرد، بصورت زیر:

```
for ( int number = 2; // initialization
      number <= 20; // loop continuation condition
      total += number, number += 2 ) // total and increment
;// empty body
```

برنامه‌نویسی ایده‌ال

گرچه ادغام عبارات بدنه **for** *با بخش سرآیند آن وجود دارد اما از انجام اینکار اجتناب کنید تا درک برنامه مشکل نشود.*

برنامه‌نویسی ایده‌ال

در صورت امکان، سایز عبارات سرآیند کنترل را محدود کنید.

برنامه:محاسبه سود

مثال بعدی در ارتباط با محاسبه یک مسئله ترکیبی با استفاده از عبارت **for** است. صورت مسئله به شرح زیر است:

شخصی 1000.00 دلار در یک حساب پس انداز با سود 5٪ سرمایه‌گذاری کرده است. با فرض اینکه کل سود نیز ذخیره می‌شود، مقدار پول موجود در حساب را در پایان هر سال در یک مدت 10 ساله محاسبه و چاپ کنید. برای بدست آوردن این مقادیر، از فرمول زیر کمک بگیرید:

$$a = p(1 + r)^n$$

در این فرمول

p میزان سرمایه اولیه

r نرخ سود

n تعداد سال‌ها

a مقدار سرمایه در پایان سال nام است.

این مسئله مستلزم حلقه‌ای است که دلالت بر انجام محاسبه‌ای برای هر سال در مدت ده سال پول باقیمانده در حساب پس‌انداز است. این راه حل در برنامه شکل ۶-۵ آورده شده است.

عبارت **for** (خطوط 35-28) مبادرت به اجرای بدنه خود به میزان 10 بار می‌کند. مقدار متغیر کنترلی از 1 تا 10 به میزان یک واحد در هر بار افزایش می‌یابد. زبان C++ حاوی عملگر توان نیست، از اینرو از تابع کتابخانه‌ای استاندارد **pow** به همین منظور استفاده کرده‌ایم (خط 31). تابع **pow(x,y)** مبادرت به محاسبه مقدار **x** به توان **y** می‌کند. در این مثال، عبارت جبری $(1+r)^n$ بصورت pow(1.0+rate,year) نوشته شده است، که متغیر **rate** نشاندهنده **r** و متغیر **year** نشاندهنده **n** است. تابع **pow** دو آرگومان از نوع **double** دریافت و یک مقدار **double** برگشت می‌دهد.

این برنامه بدون سرآیند فایل <**cmath**> کامپایل نمی‌شود. تابع **pow** نیازمند دو آرگومان از نوع **double** است. دقت کنید که متغیر **year** از نوع صحیح است. سرآیند <**cmath**> حاوی اطلاعاتی است که به کامپایلر می‌گوید تا مقدار **year** را بطور موقت تبدیل به نوع **double** کند، قبل از اینکه تابع فراخوانی شود. این اطلاعات در نمونه اولیه تابع **pow** وجود دارند. در فصل ششم با چندین تابع کتابخانه **math** آشنا خواهید شد.

```
1  // Fig. 5.6: fig05_06.cpp
2  // Compound interest calculations with for.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6  using std::fixed;
7
8  #include <iomanip>
9  using std::setw; // enables program to set a field width
10 using std::setprecision;
11
12 #include <cmath> // standard C++ math library
13 using std::pow; // enables program to use function pow
```

```
14
15 int main()
16 {
17    double amount; // amount on deposit at end of each year
18    double principal = 1000.0; // initial amount before interest
19    double rate = .05; // interest rate
20
21    // display headers
22    cout << "Year" << setw( 21 ) << "Amount on deposit" << endl;
23
24    // set floating-point number format
25    cout << fixed << setprecision( 2 );
26
27    // calculate amount on deposit for each of ten years
28    for ( int year = 1; year <= 10; year++ )
29    {
30       // calculate new amount for specified year
31       amount = principal * pow( 1.0 + rate, year );
32
33       // display the year and the amount
34       cout << setw( 4 ) << year << setw( 21 ) << amount << endl;
35    } // end for
36
37    return 0; // indicate successful termination
38 } // end main
```

```
Year     Amount on deposit
1                  1050.00
2                  1102.50
3                  1157.63
4                  1215.51
5                  1276.28
6                  1340.10
7                  1407.10
8                  1477.46
9                  1551.33
10                 1628.89
```

شکل ٦-٥ | عبارت for برای محاسبه سود سرمایه‌گذاری.

خطای برنامه‌نویسی

بطور کلی، فراموش کردن الحاق فرآیند سرآیند مورد نیاز به هنگام استفاده از توابع کتابخانه استاندارد (همانند <cmatch>) خطای کامپایل بدنبال خواهد داشت.

دقت به هنگام استفاده از نوع double در محاسبات مالی

دقت کنید که متغیرهای **amount** و **principal** و **rate** از نوع ***double*** هستند. به این دلیل از این نوع استفاده کرده‌ایم که می‌خواهیم به بخش‌های کسری رسیدگی کرده و به نوعی نیاز داریم که امکان انجام محاسبات دقیق در زمینه مسائل مالی را فراهم آوریم. متاسفانه این نوع می‌تواند مشکل‌ساز شود. در این بخش با یک توضیح ساده شاهد خواهید بود که چگونه به هنگام استفاده از نوع **float** یا **double** در نمایش‌های مالی می‌توانیم دچار اشتباه شویم (فرض کنید از **setprecision(2)** برای مشخص کردن دو رقم دقت به هنگام چاپ استفاده کرده‌ایم): دو مبلغ دلاری ذخیره شده در ماشین می‌تواند 14.234 (که 14.23 چاپ می‌شود) و 18.673 (که 18.67 چاپ خواهد شد) را تولید کند. زمانیکه این مبالغ با هم جمع

شوند، مجموع داخلی 32.907 تولید می‌شود که بصورت 32.91 چاپ می‌گردد. از اینرو نتیجه چاپی به صورت زیر ظاهر خواهد شد.

```
  14.23
+ 18.67
-------
  32.91
```

اما در صورتیکه خودمان این اعداد را با هم جمع کنیم مجموع 32.90 را بدست خواهیم آورد. پس مشکلی در کار است.

برنامه‌نویسی ایده‌ال

*از متغیرهای نوع **float** یا **double** برای انجام محاسبات مالی استفاده نکنید. عدم دقت کافی در چنین اعدادی می‌تواند در نتیجه تولیدی از محاسبات مالی، شما را دچار مشکل کند.*

کنترل کننده‌های جریان برای قالب‌بندی کردن خروجی عددی

عبارت خروجی در خط 25 قبل از حلقه **for** و عبارت خروجی در خط 34 در حلقه **for** ترکیب شده‌اند تا مقادیر متغیرهای **year** و **amount** با قالب‌بندی (فرمت) تصریح شده توسط کنترل‌کننده‌های جریان پارامتری شده **setprecision** و **setw** و کنترل‌کننده استریم غیرپارامتری **fixed** چاپ شوند. کنترل‌کننده استریم **setw(4)** مشخص می‌کند که باید مقدار خروجی بعدی به طول فیلد مشخص شده یعنی 4 به نمایش درآید، یعنی، **cout** مقدار را با حداقل 4 موقعیت کاراکتری چاپ خواهد کرد. اگر مقداری که چاپ می‌شود، کمتر از 4 کاراکتر طول داشته باشد، مقدار از سمت راست تراز می‌شود (بطور پیش‌فرض). اگر مقدار خروجی بیش از 4 کاراکتر طول داشته باشد، طول فیلد گسترش می‌یابد تا به کل مقدار جا دهد. برای تاکید بر این نکته که مقادیر در خروجی باید از سمت چپ تراز شوند، کافیست از کنترل‌کننده استریم غیرپارامتری **left** استفاده شود (در سرآیند <**iostream**> قرار دارد). ترازبندی از سمت راست را می‌توان با استفاده از کنترل‌کننده استریم غیرپارامتری **right** بدست آورد.

قالب‌بندی‌های دیگر، در عبارات خروجی بر این نکته دلالت می‌کنند که متغیر **amount** بصورت یک مقدار با نقطه ثابت با یک نقطه دیسمال (تصریح شده در خط 25 با کنترل‌کننده استریم **fixed**) تراز از سمت راست در فیلدی به میزان 21 کاراکتر (تصریح شده در خط 34 با **setw(21)**) و با دقت دو رقم در سمت راست نقطه دیسمال (تصریح شده در خط 25 با کنترل‌کننده **setprecision(2)**) چاپ خواهد شد. کنترل‌کننده‌های استریم **fixed** و **setprecision** را بر روی استریم خروجی (یعنی **cout**) و قبل از حلقه **for** اعمال کرده‌ایم چرا که این تنظیمات قالب‌بندی تا زمانیکه تغییر نیافته‌اند ثابت باقی می‌مانند، به چنین تنظیماتی، تنظیمات *چسبنده* می‌گویند. از اینرو، نیازی ندارند در هر بار تکرار حلقه بکار گرفته شوند. با

این همه، طول فیلد (میدان) مشخص شده با **setw** فقط بر مقدار بعدی در خروجی بکار گرفته می‌شود. در فصل پانزدهم در ارتباط با قابلیت‌های قالبندی ورودی/خروجی زبان C++ صحبت خواهیم کرد.

به عبارت **rate + 1.0** که بصورت یک آرگومان در تابع **pow** و در بدنه عبارت **for** قرار دارد، توجه کنید. در واقع، این محاسبه همان نتیجه را در زمان هر تکرار حلقه تولید می‌کند، از اینرو تکرار بی‌موردی صورت گرفته و باید این عبارت یکبار و قبل از حلقه محاسبه شود.

کارایی

برخی از کامپایلرها حاوی ویژگی‌های بهینه‌سازی هستند که سبب افزایش کارایی کد نوشته شده توسط شما می‌شوند، اما بهتر است از همان مرحله شروع کار کدنویسی، کدهای خود را کارآمد بنویسیم.

کارایی

از قراردادن عبارات محاسباتی در داخل حلقه که مقدار آنها در هر بار اجرای حلقه تغییری نمی‌یابد خودداری کنید. چنین عباراتی فقط یکبار و قبل از حلقه باید ارزیابی شوند.

۵-۵ عبارت تکرار do...while

عملکرد عبارت تکرار **do...while** همانند عبارت **while** است. در عبارت **while**، شرط تکرار حلقه در ابتدای حلقه تست می‌شود، قبل از اینکه بدنه حلقه به اجرا درآید. عبارت **do...while** شرط تکرار حلقه را پس از اجرای حلقه تست می‌کند. از اینرو، در یک عبارت **do...while**، همیشه بدنه حلقه حداقل یکبار به اجرا در می‌آید. اگر فقط یک عبارت در بدنه وجود داشته باشد، استفاده از براکت‌ها در **do...while** ضرورتی ندارد. با این همه، معمولا برای اجتناب از سردرگمی مابین عبارات **while** و **do...while** از براکت‌ها استفاده می‌شود. برای مثال،

```
while (شرط)
```

نشان‌دهنده سرآیند عبارت **while** است. یک عبارت **do...while** بدون براکت‌ها در اطراف بدنه خود (با یک عبارت) بصورت زیر خواهد بود

```
do
   عبارت
while ( شرط );
```

که می‌تواند باعث اشتباه شود. خط آخر، می‌تواند توسط خواننده به غلط بعنوان یک عبارت while با عبارت تهی تفسیر شود (وجود سیمکولن در انتهای شرط). از اینرو، برای اجتناب از اشتباه، بهتر است در یک عبارت **do...while** با یک عبارت از براکت‌ها استفاده شود:

```
do {
```

```
        عبارت;
    } while ( شرط );
```

در برنامه شکل ۷-۵ از یک عبارت **do...while** برای چاپ مقادیر از 10 تا 1 استفاده شده است. عبارت **do...while** در خطوط 15-11 اعمال شده است. در اولین برخورد برنامه با این عبارت، خط 13 اجرا شده و مقدار متغیر **counter** (با مقدار 1) چاپ شده، سپس **counter** یک واحد افزایش می‌یابد (خط 14). سپس شرط در خط 15 ارزیابی می‌شود. اکنون متغیر **counter** حاوی مقدار **2** است که کمتر یا مساوی با **10** می‌باشد، چون شرط برقرار است، عبارت **do...while** مجدداً اجرا می‌شود. در بار دهم که عبارت اجرا شد، عبارت خط 13 مقدار **10** را چاپ کرده و در خط 14، مقدار **counter** به **11** افزایش می‌یابد. در این لحظه، شرط موجود در خط 15، نادرست ارزیابی شده و برنامه از عبارت **do...while** خارج می‌شود و برنامه به سراغ عبارت پس از حلقه می‌رود (خط 17).

```
1  // Fig. 5.7: fig05_07.cpp
2  // do...while repetition statement.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     int counter = 1; // initialize counter
10
11    do
12    {
13       cout << counter << " "; // display counter
14       counter++; // increment counter
15    } while ( counter <= 10 ); // end do...while
16
17    cout << endl; // output a newline
18    return 0; // indicate successful termination
19 } // end main
```

```
1 2 3 4 5 6 7 8 9 10
```

شکل ۷-۵ | عبارت تکرار do...while

دیاگرام فعالیت UML عبارت do...while

شکل ۸-۵ حاوی دیاگرام فعالیت UML برای عبارت **do...while** است. این دیاگرام به وضوح نشان می‌دهد که شرط تکرار حلقه حداقل پس از یک بار اجرای عبارات موجود در بدنه حلقه، ارزیابی نخواهد شد. این دیاگرام فعالیت را با عبارت **while** بکار رفته در شکل ۶-۴ مقایسه کنید. مجدداً، توجه کنید که این دیاگرام حاوی یک نماد وضعیت عمل و تصمیم‌گیری است.

شکل ۸-۵ | دیاگرام فعالیت UML عبارت تکرار do...while

۵-۶ عبارت چند انتخابی switch

در فصل گذشته، در ارتباط با عبارت تک انتخابی **if** و دو انتخابی **if...else** صحبت کردیم. زبان C++ دارای عبارت چند انتخابی **switch** برای رسیدگی به چنین شرایطی است.

کلاس GradeBook با عبارت switch

در مثال بعدی، مبادرت به عرضه یک نسخه بهبود یافته از کلاس **GradeBook** معرفی شده در فصل سوم و فصل چهارم می‌کنیم. نسخه جدید از کاربر می‌خواهد تا مجموعه‌ای از امتیازات حرفی را وارد کرده، سپس تعداد دانشجویانی که هر کدام امتیازی دریافت کرده‌اند، به نمایش در می‌آورد. این کلاس از یک عبارت **switch** برای تعیین اینکه امتیاز وارد شده یک A، B، C، D یا F می‌باشد و افزایش مقدار شمارنده امتیاز وارد شده، استفاده می‌کند. کلاس **GradeBook** در برنامه شکل ۹-۵ تعریف شده و تعریف تابع عضو آن در برنامه شکل ۱۰-۵ قرار دارد. شکل ۱۱-۵ نمایشی از اجرای نمونه برنامه همراه ورودی‌ها و خروجی برنامه **main** است که از کلاس **GradeBook** برای پردازش امتیازات استفاده می‌کند. همانند نسخه‌های قبلی تعریف کلاس، تعریف کلاس **GradeBook** در شکل ۹-۵ حاوی نمونه اولیه تابع برای توابع عضو **setCourseName** در خط 13، **getCourseName** در خط 14 و **displayMessage** در خط 15 به همراه سازنده کلاس در خط 12 است. همچنین در تعریف کلاس عضو داده **courseName** بصورت ***private*** اعلان شده است (خط 19).

کلاس **GradeBook** در شکل۹-۵ دارای پنج عضو داده **private** است (خطوط 14-20)، متغیرهای شمارنده برای هر امتیاز (یعنی برای A، B، C، D و F). همچنین کلاس دارای دو تابع عضو ***public*** بنام‌های **inputGrades** و **displayGradeReport** است. تابع عضو **inputGrade** (اعلان شده در خط 16) مبادرت خواندن حروف امتیازی به تعداد اختیاری از طرف کاربر با روش مقدار مراقبتی می‌کند و شمارنده امتیاز مقتضی را برای هر امتیاز وارد شده به روز می‌نماید. تابع عضو **displayGradeReport** (اعلان شده در خط 17) گزارشی از تعداد دانشجویان دریافت‌کننده هر امتیاز به نمایش در می‌آورد.

```
1  // Fig. 5.9: GradeBook.h
2  // Definition of class GradeBook that counts A, B, C, D and F grades.
3  // Member functions are defined in GradeBook.cpp
4
5  #include <string> // program uses C++ standard string class
6  using std::string;
7
8  // GradeBook class definition
9  class GradeBook
10 {
11 public:
12    GradeBook( string ); // constructor initializes course name
13    void setCourseName( string ); // function to set the course name
14    string getCourseName(); // function to retrieve the course name
15    void displayMessage(); // display a welcome message
16    void inputGrades(); // input arbitrary number of grades from user
17    void displayGradeReport(); // display a report based on the grades
18 private:
19    string courseName; // course name for this GradeBook
20    int aCount; // count of A grades
21    int bCount; // count of B grades
22    int cCount; // count of C grades
23    int dCount; // count of D grades
24    int fCount; // count of F grades
```

```
25 }; // end class GradeBook
```

شکل ۹-۵ | تعریف کلاس GradeBook.

فایل کد منبع **GradeBook.cpp** در شکل ۱۰-۵ حاوی تعریف تابع عضو کلاس **GradeBook** است. توجه کنید زمانیکه یک شی **GradeBook** برای اولین بار ایجاد می‌شود و هنوز هیچ امتیازی وارد نشده است، خطوط 20-16 در سازنده مبادرت به مقداردهی اولیه پنج شمارنده امتیاز با صفر می‌کنند. همانطوری که بزودی خواهید دید، این شمارنده‌ها در تابع عضو **inputGrade** و بعنوان امتیازهای ورودی کاربر افزایش می‌یابند. تعاریف توابع عضو **setCourseName**، **getCourseName** و **displayMessage** با نسخه‌های قبلی موجود در کلاس **GradeBook** یکسان هستند. اجازه دهید تا نگاهی به توابع عضو جدید در **GradeBook** داشته باشیم.

```
1  // Fig. 5.10: GradeBook.cpp
2  // Member-function definitions for class GradeBook that
3  // uses a switch statement to count A, B, C, D and F grades.
4  #include <iostream>
5  using std::cout;
6  using std::cin;
7  using std::endl;
8
9  #include "GradeBook.h" // include definition of class GradeBook
10
11 // constructor initializes courseName with string supplied as argument;
12 // initializes counter data members to 0
13 GradeBook::GradeBook( string name )
14 {
15    setCourseName( name ); // validate and store courseName
16    aCount = 0; // initialize count of A grades to 0
17    bCount = 0; // initialize count of B grades to 0
18    cCount = 0; // initialize count of C grades to 0
19    dCount = 0; // initialize count of D grades to 0
20    fCount = 0; // initialize count of F grades to 0
21 } // end GradeBook constructor
22
23 // function to set the course name; limits name to 25 or fewer characters
24 void GradeBook::setCourseName( string name )
25 {
26    if ( name.length() <= 25 ) // if name has 25 or fewer characters
27       courseName = name; // store the course name in the object
28    else // if name is longer than 25 characters
29   { // set courseName to first 25 characters of parameter name
30       courseName = name.substr( 0, 25 ); // select first 25 characters
31       cout << "Name \"" << name << "\" exceeds maximum length (25).\n"
32          << "Limiting courseName to first 25 characters.\n" << endl;
33    } // end if...else
34 } // end function setCourseName
35
36 // function to retrieve the course name
37 string GradeBook::getCourseName()
38 {
39    return courseName;
40 } // end function getCourseName
41
42 // display a welcome message to the GradeBook user
43 void GradeBook::displayMessage()
44 {
45    // this statement calls getCourseName to get the
46    // name of the course this GradeBook represents
47    cout << "Welcome to the grade book for\n" << getCourseName() << "!\n"
48       << endl;
```

```
49 } // end function displayMessage
50
51 // input arbitrary number of grades from user; update grade counter
52 void GradeBook::inputGrades()
53 {
54    int grade; // grade entered by user
55
56    cout << "Enter the letter grades." << endl
57       << "Enter the EOF character to end input." << endl;
58
59    // loop until user types end-of-file key sequence
60    while ( ( grade = cin.get() ) != EOF )
61    {
62      // determine which grade was entered
63       switch ( grade ) // switch statement nested in while
64       {
65          case 'A': // grade was uppercase A
66          case 'a': // or lowercase a
67             aCount++; // increment aCount
68             break; // necessary to exit switch
69
70          case 'B': // grade was uppercase B
71          case 'b': // or lowercase b
72             bCount++; // increment bCount
73             break; // exit switch
74
75          case 'C': // grade was uppercase C
76         case 'c': // or lowercase c
77             cCount++; // increment cCount
78             break; // exit switch
79
80          case 'D': // grade was uppercase D
81          case 'd': // or lowercase d
82             dCount++; // increment dCount
83             break; // exit switch
84
85          case 'F': // grade was uppercase F
86          case 'f': // or lowercase f
87             fCount++; // increment fCount
88             break; // exit switch
89
90          case '\n': // ignore newlines,
91          case '\t': // tabs,
92          case ' ': // and spaces in input
93             break; // exit switch
94
95          default: // catch all other characters
96             cout << "Incorrect letter grade entered."
97                << " Enter a new grade." << endl;
98             break; // optional; will exit switch anyway
99       } // end switch
100    } // end while
101 } // end function inputGrades
102
103 // display a report based on the grades entered by user
104 void GradeBook::displayGradeReport()
105 {
106    // output summary of results
107    cout << "\n\nNumber of students who received each letter grade:"
108       << "\nA: " << aCount // display number of A grades
109       << "\nB: " << bCount // display number of B grades
110       << "\nC: " << cCount // display number of C grades
111       << "\nD: " << dCount // display number of D grades
112       << "\nF: " << fCount // display number of F grades
113       << endl;
114 } // end function displayGradeReport
```

شکل ۱۰-۵ | کلاس GradeBook از عبارت switch برای شمارش امتیازات A, B, C, D و F استفاده می کند.

خواندن کاراکترهای ورودی

کاربر مبادرت به وارد کردن امتیازات حرفی برای یک واحد درسی در تابع **inputGrades** می‌کند (خطوط 101-52). در سرآیند حلقه **while** در خط 60، ابتدا عبارت تخصیصی قرار گرفته در درون پرانتز (**grade = cin.get()**) اجرا می‌شود. تابع **cin.get()** یک کاراکتر از صفحه کلید خوانده و آن را در متغیر **grade** از نوع صحیح ذخیره می‌سازد (اعلان شده در خط 54). معمولاً کاراکترها در متغیرهای از نوع **char** ذخیره می‌شوند، با این همه، می‌توان کاراکترها را در هر نوع داده صحیحی ذخیره کرد، چرا که نشاندهنده 1 بایت صحیح در کامپیوتر هستند. از اینرو، می‌توانیم براساس استفاده، با یک کاراکتر همانند یک مقدار صحیح یا بعنوان یک کاراکتر رفتار کنیم. برای مثال، عبارت

```
cout << "The character (" << 'a' << ") has the value"
     << static_cast< int > ( 'a' ) << endl;
```

مبادرت به چاپ کاراکتر **a** و مقدار صحیح آن بصورت زیر می‌کند:

```
The character (a) has the value 97
```

عدد صحیح **97** نشاندهنده شماره عددی کاراکتر **a** در کامپیوتر است. اکثر کامپیوترها از مجموعه کاراکتری (*American Standard Code for Information Interchange*) ASCII استفاده می‌کنند، که در این مورد 97 نشاندهنده حرف کوچک 'a' است.

بطور کلی عبارات تخصیص‌دهنده مبادرت به تخصیص مقدار قرار گرفته در سمت راست به متغیر قرار گرفته در سمت چپ علامت = می‌کنند. بنابر این مقدار عبارت تخصیصی **grade=cin.get()** همان مقدار برگشتی از سوی **cin.get()** بوده و به متغیر **grade** تخصیص می‌یابد. می‌توان از عبارات تخصیص‌دهنده برای تخصیص یک مقدار به چندین متغیر استفاده کرد. برای مثال در عبارت زیر

```
a=b=c=0;
```

ابتدا تخصیص **c=0** صورت می‌گیرد چرا که عملگر = دارای شرکت‌پذیری از سمت راست به چپ است. سپس به متغیر **b** مقدار تخصیصی **c=0**، تخصیص می‌یابد (که صفر است). سپس متغیر **a** حاوی مقدار تخصیصی **b=(c=0)** خواهد شد که آن هم صفر است. در برنامه، مقدار عبارت تخصیصی **grade=cin.get()** با مقدار **EOF** مقایسه می‌شود (نمادی که کوتاه شده عبارت "*end-of-file*" است). ما از EOF (که معمولاً دارای مقدار **-1** است) بعنوان مقدار مراقبتی استفاده کرده‌ایم. با این همه، نیازی به تایپ مقدار -1 یا تایپ حروف EOF بعنوان مقدار مراقبتی ندارید. بجای آن از یک ترکیب کلیدی استفاده کرده‌ایم، که نشاندهنده EOF در سیستم است. **EOF** یک ثابت سمبولیک سیستم است که در سرآیند فایل **<iostream>** تعریف شده است. اگر مقداری به **grade** تخصیص دهید که معادل با EOF

باشد، حلقه **while** در خطوط 100-60 بکار خود خاتمه می‌دهد. نمایش کاراکترهای وارد شده به این برنامه را بصورت مقادیر صحیح انتخاب کرده‌ایم، چرا که EOF دارای یک مقدار صحیح است.

در سیستم‌های UNIX/Linux و برخی از سیستم‌های دیگر، EOF با تایپ

```
<ctrl> d
```

در یک خط وارد می‌شود. این عبارت به این معنی است که کلید *ctrl* را فشار و پایین نگه داشته و سپس کلید d فشار داده شود. در سیستم‌های دیگر همانند Microsoft Windows، می‌توان EOF را با تایپ

```
<ctrl> z
```

وارد کرد.

قابلیت حمل

کلیدهای ترکیبی برای وارد کردن EOF به سیستم وابسته هستند.

در این برنامه، کاربر امتیازها را توسط صفحه کلید وارد می‌کند. زمانیکه کاربر کلید Enter را فشار دهد، کاراکترها توسط تابع **cin.get()** خوانده می‌شوند، یک کاراکتر در یک زمان. اگر کاراکتر وارد شده EOF نباشد، برنامه وارد عبارت **switch** می‌شود (خط 99-63) که شمارنده امتیاز را متناسب با حرف وارد شده، یک واحد افزایش می‌دهد.

جزئیات عبارت switch

عبارت **switch** متشکل از تعدادی برچسب **case** و یک حالت **default** اختیاری است. از این عبارت در این مثال برای تعیین اینکه کدام شمارنده باید براساس امتیاز وارد شده افزایش یابد، استفاده شده است. زمانیکه کنترل برنامه به **switch** می‌رسد، برنامه مبادرت به ارزیابی عبارت موجود در درون پرانتزها می‌کند (یعنی **grade**) که بدنبال کلمه کلیدی **switch** قرار گرفته است (خط 63). به این عبارت، *عبارت کنترلی* گفته می‌شود. عبارت **switch** شروع به مقایسه مقدار عبارت کنترلی با هر برچسب **case** می‌کند. فرض کنید که کاربر حرف C را بعنوان کد امتیاز وارد کرده باشد. برنامه شروع به مقایسه C با هر **case** موجود در بدنه **switch** می‌کند. اگر مطابقتی یافت شود (در خط 75، **case 'C'**)، برنامه عبارات موجود در آن **case** را به اجرا در می‌آورد. در ارتباط با حرف C، خط 77 مبادرت به افزایش **cCount** به می‌زان یک واحد می‌کند. عبارت **break** (خط 78) سبب می‌شود که کنترل برنامه به اولین عبارت پس از **switch** منتقل شود که در این برنامه کنترل به خط 100 انتقال می‌یابد. این خط، انتهای بدنه حلقه **while** را نشان می‌دهد (خطوط 100-60)، از اینرو جریان کنترل به سمت شرط **while** در خط 60 می‌رود تا تعیین نماید که آیا حلقه بایستی ادامه یابد یا خیر.

caseهای موجود در عبارت **switch** بطور صریح مبادرت به تست حروف کوچک و بزرگ حرف‌های A، B، C، D و F می‌کنند. توجه کنید که caseهای موجود در خطوط 66-65 در تست مقادیر 'A' و 'a' کاربرد دارند (هر دو نشاندهنده امتیاز A می‌باشند). لیست caseها در این روش بصورت متوالی بوده و عبارتی مابین آنها وجود ندارد و به هر دو **case** اجازه می‌دهد تا یک مجموعه از عبارات را به اجرا درآورند، زمانیکه عبارت کنترلی با 'A' یا 'a' ارزیابی شود، عبارات قرار گرفته در خطوط 68-67 اجرا خواهند شد. توجه کنید که هر **case** می‌تواند چندین عبارت داشته باشد. عبارت انتخابی **switch** متفاوت از دیگر عبارات کنترلی بوده و نیازی به استفاده از براکت‌ها در اطراف چندین عبارت در هر **case** ندارد.

بدون عبارات **break**، هر زمان که مطابقتی در **switch** تشخیص داده شود، عبارات آن **case** و caseهای متعاقب آن اجرا خواهند شد تا اینکه به یک عبارت **break** یا به انتهای **switch** برسد. به این حالت "*falling through*" یا عدم دستیابی به نتیجه در caseهای بعدی گفته می‌شود.

خطای برنامه‌نویسی

*نتیجه فراموش کردن عبارت **break** در مکانی که به آن در **switch** نیاز است، یک خطای منطقی است.*

خطای برنامه‌نویسی

*حذف فاصله مابین کلمه **case** و مقدار ارزش در یک عبارت **switch** خطای منطقی است. برای مثال، نوشتن **:case3** بجای **case 3:** یک برچسب بلااستفاده بوجود می‌آورد.*

تدارک دیدن حالت default

اگر هیچ مطابقتی مابین مقدار عبارت کنترلی و یک برچسب **case** پیدا نشود، حالت **default** اجرا خواهد شد (خطوط 98-95). در این مثال از حالت **default** برای پردازش تمام مقادیر عبارت کنترلی که امتیازهای معتبر نیستند، کاراکترهای خط جدید، تب یا فاصله استفاده کرده‌ایم. اگر هیچ مطابقتی رخ ندهد، حالت **default** اجرا می‌شود و خطوط 97-96 یک پیغام خطا به نمایش در می‌آورند تا بر این نکته دلالت کنند که یک حرف امتیازی اشتباه وارد شده است. اگر هیچ مطابقتی در یک عبارت **switch** رخ ندهد و این عبارت فاقد حالت **default** باشد، کنترل برنامه بسادگی به اجرای اولین عبارت پس از **switch** ادامه خواهد داد.

برنامه‌نویسی ایده‌ال

*در عبارات **switch** حالت **default** را در نظر بگیرید. در حالت **default** برنامه‌نویس می‌تواند مواردی که برای پردازش موارد استثناء پیش می‌آیند در نظر بگیرد. با اینکه می‌توان به هر ترتیبی شرط‌های **case** و حالت **default** را در یک عبارت **switch** قرار داد، اما بهتر است ضابطه **default** در آخر قرار داده شود.*

برنامه‌نویسی ایده‌ال

*در یک عبارت **switch** که **default** در انتهای آن قرار دارد، نیازی نیست که ضابطه **default** حاوی دستور*

break باشد. برخی از برنامه‌نویسان به منظور حفظ تقارن و وضوح بیشتر با سایر caseها ترجیح می‌دهند از break در default استفاده کنند.

نادیده گرفتن کارکترهای خط جدید،تب و فاصله در ورودی

خطوط 93-90 در عبارت **switch** شکل ۱۰-۵ سبب می‌شوند تا برنامه کاراکترهای خط جدید، تب و فاصله‌ها را در نظر نگیرد. خواندن یک کاراکتر در هر زمان می‌تواند مشکل‌ساز شود. برای داشتن برنامه‌ای که کاراکترها را بخواند، بایستی آنها را به کامپیوتر با فشردن کلید Enter صفحه‌کلید ارسال کنیم. با اینکار یک کاراکتر خط جدید (*newline*) در ورودی پس از کاراکترهایی که مایل به پردازش آنها هستیم، وارد می‌شود. غالباً برای اینکه برنامه بدرستی کار کند نیاز است تا به این کاراکتر رسیدگی شود. با قرار دادن **case** سابق‌الذکر در عبارت **switch**، مانع از نمایش پیغام خطا در حالت **default** در هر بار مواجه شدن با کاراکترهای خط جدید، تب (tab) یا فاصله در ورودی شده‌ایم.

خطای برنامه‌نویسی

عدم پردازش کاراکترهای خط جدید و سایر کاراکترهای white-space در ورودی، زمانیکه در هر بار یک کاراکتر خوانده می‌شود، می‌تواند شما را با خطاهای منطقی مواجه سازد.

تست کلاس GradeBook

برنامه شکل ۱۱-۵ یک شی **GradeBook** ایجاد می‌کند (خط 9). خط 11 تابع عضو **displayMessage** شی را برای نمایش پیغام خوش‌آمدگویی به کاربر فراخوانی می‌کند. خط 12 تابع عضو **inputGrades** را برای خواندن مجموعه‌ای از امتیازها از سوی کاربر و شمارش تعداد دانشجویان دریافت‌کننده امتیاز فراخوانی می‌کند. به پنجره خروجی/ورودی به نمایش درآمده در شکل ۱۱-۵ توجه کنید که یک پیغام خطا در واکنش به وارد کردن یک امتیاز اشتباه (یعنی E) به نمایش درآورده است. خط 13 مبادرت به فراخوانی تابع عضو **displayGradeReport** تعریف شده در خطوط 114-104 از شکل ۱۰-۵ می‌کند و این تابع نتیجه برنامه را براساس امتیازهای وارد شده به نمایش در می‌آورد.

```
1  // Fig. 5.11: fig05_11.cpp
2  // Create GradeBook object, input grades and display grade report.
3
4  #include "GradeBook.h" // include definition of class GradeBook
5
6  int main()
7  {
8     // create GradeBook object
9     GradeBook myGradeBook( "CS101 C++ Programming" );
10
11    myGradeBook.displayMessage(); // display welcome message
12    myGradeBook.inputGrades(); // read grades from user
13    myGradeBook.displayGradeReport(); // display report based on grades
14    return 0; // indicate successful termination
15 } // end main
```

```
Welcome to the grade book for
CS101 C++ Programming!

Enter a letter grades.
Enter the EOF character to end input.
a
B
c
C
A
d
f
C
E
Incorrect letter grade entered. Enter a new grade.
D
A
b
^z

Number of student who received each letter grade:
A:3
B:2
C:3
D:2
F:1
```

شکل ۱۱-۵ | ایجاد یک شی GradeBook و فراخوانی توابع عضو آن.

دیاگرام فعالیت UML عبارت switch

شکل ۱۲-۵ نشاندهنده دیاگرام فعالیت UML یک عبارت چند انتخابی **switch** است. اکثر عبارات **switch** از یک **break** در هر **case** استفاده می‌کنند تا به عبارت **switch** پس از پردازش آن **case** خاتمه دهند. شکل۱۲-۵ بر استفاده از عبارت **break** در دیاگرام فعالیت تاکید دارد. بدون عبارت **break**کنترل به اولین عبارت پس از ساختار **switch** پس از اینکه یک **case** پردازش شد، منتقل نمی‌شود. بجای آن کنترل به سراغ اجرای **case** بعدی می‌رود.

این دیاگرام به وضوح نشان می‌دهد که عبارت **break** در انتهای یک **case** سبب انتقال بلافاصله کنترل به خارج از عبارت **switch** می‌شود. مجدداً توجه کنید که این دیاگرام حاوی نمادهای وضعیت عمل و تصمیم‌گیری است. همچنین توجه کنید که در این دیاگرام از نمادهای ادغام برای انتقال از عبارات **break** به وضعیت پایانی استفاده شده است.

به هنگام استفاده از عبارت **switch**، بخاطر داشته باشید که می‌توان از آن فقط برای تست یک مقدار ارزشی ثابت استفاده کرد. هر ترکیبی از ثابت‌های کاراکتری و ثابت‌های عددی صحیح، بصورت یک ثابت عددی صحیح ارزیابی می‌شوند. یک ثابت کاراکتری بصورت یک کاراکتر خاص در میان علامت نقل قول همانند **'A'** است. یک ثابت عددی، یک مقدار عددی صحیح است. همچنین هر برچسب **case** می‌تواند تعیین‌کننده یک عبارت ارزشی ثابت باشد.

خطای برنامه‌نویسی

مشخص کردن یک عبارت همراه با متغیرها همانند ***a+b*** در برچسب ***case*** یک ***switch***، خطای نحوی

است.

خطای برنامه‌نویسی

*تدارک دیدن برچسب‌های یکسان در یک عبارت **switch** خطای کامپایل بدنبال خواهد داشت. همچنین قرار دادن عبارات مختلف در caseها که مقدار آنها یکسان ارزیابی گردند نیز خطای کامپایل بوجود می‌آورد. برای مثال قرار دادن **case4+1:** و **case3+2:** در یک عبارت **switch** خطای کامپایل است چرا که هر دو آنها برابر **case 5:** هستند.*

شکل ۱۲-۵ | دیاگرام فعالیت UML ساختار switch با عبارت break.

تکاتی در ارتباط با نوع داده

زبان C++ دارای نوع داده با سایزهای مختلف است. برای مثال، امکان دارد برنامه‌های مختلف، به اعداد صحیح با سایزهای متفاوت نیاز داشته باشند. C++ دارای چندین نوع داده برای عرضه مقادیر صحیح است. طول مقادیر صحیح برای هر نوع بستگی به سخت‌افزار کامپیوتر دارد. علاوه بر نوع‌های **int** و **char**، زبان C++ نوع‌های **short** (کوتاه شده short int) و **long** (کوتاه شده long int) را در نظر گرفته است. حداقل محدودهٔ مقادیر صحیح از نوع **short** از 32.768- تا 32.767 می‌باشد. برای اکثر محاسبات صحیح، مقادیر صحیح از نوع **long** کافی هستند. حداقل محدودهٔ مقادیر از نوع **long** از 2.147.483.648- تا 2.147.483.647 است. بر روی اکثر کامپیوترها، مقادیر **int** معادل **short** یا **long** هستند. محدودهٔ مقادیر برای یک **int** حداقل برابر با مقادیر **short** بوده و بیشتر از مقادیر **long** نمی‌باشند. از نوع داده **char** می‌توان برای عرضه هر کاراکتری در مجموعه کاراکتری کامپیوتر استفاده کرد. همچنین می‌توان از آن برای نمایش مقادیر صحیح کوچک استفاده کرد.

قابلیت حمل

*بدلیل اینکه intها می‌توانند مابین سیستم‌ها سایز متفاوتی داشته باشند، اگر انتظار دارید محاسبه‌ای سبب تولید مقداری بیش از محدودهٔ 32.767 تا 32.763- نماید از نوع **long** استفاده کرده و در صورت امکان برنامه را بر روی سیستم‌های مختلف اجرا کنید.*

کارایی

اگر استفاده بهینه از حافظه مطرح باشد، می‌توانید از مقادیر صحیح با سایز کوچک استفاده کنید.

۵-۷ عبارات break و continue

علاوه بر عبارات کنترلی و انتخابی، زبان C++ عبارات **break** و **continue** را برای ایجاد تغییر در جریان کنترل در نظر گرفته است. بخش بعدی شما را با نحوه استفاده از **break** در خاتمه دادن به اجرای یک عبارت **switch** آشنا خواهد کرد. همچنین این بخش در مورد نحوه استفاده از **break** در یک عبارت تکرار مطالبی عرضه خواهد کرد.

عبارت break

دستور **break** با اجرا در عبارات **while ,for ,do..while** یا **switch** سبب می‌شود تا کنترل بلافاصله از عبارت خارج شده، و برنامه با اولین عبارت پس از عبارت ادامه می‌یابد. معمولا از دستور **break** برای خارج شدن زود هنگام از حلقه یا پرش از مابقی عبارت **switch** (همانند برنامه ۱۰-۵) استفاده می‌شود. برنامه شکل ۱۳-۵ به توضیح عملکرد دستور **break** در عبارت تکرار **for** پرداخته است (خط 14).

```
1  // Fig. 5.13: fig05_13.cpp
2  // break statement exiting a for statement.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     int count; // control variable also used after loop terminates
10
11    for ( count = 1; count <= 10; count++ ) // loop 10 times
12    {
13       if ( count == 5 ) // if count is 5,
14          break;         // terminate loop
15
16       cout << count << " ";
17    } // end for
18
19    cout << "\nBroke out of loop at count = " << count << endl;
20    return 0; // indicate successful termination
21 } // end main
```

```
1 2 3 4
Broke out of loop at count = 5
```

۱۳-۵ | عبارت break در یک عبارت for.

زمانیکه عبارت **if** تشخیص می‌دهد که **count** برابر 5 است، دستور **break** اجرا می‌شود. با اینکار عبارت **for** خاتمه می‌یابد و برنامه به خط 19 منتقل می‌شود (بلافاصله پس از عبارت **for**)، و پیغامی مبنی بر اینکه متغیر کنترلی به هنگام خاتمه حلقه چه مقداری داشته به نمایش در می‌آید. عبارت **for** بطور کامل و فقط چهار بار بجای ده بار اجرا می‌شود. توجه کنید که متغیر کنترلی **count** در خارج از سرآیند عبارت **for** تعریف شده است، از اینروست که می‌توانیم از متغیر کنترلی هم در بدنه حلقه و هم پس از آن استفاده کنیم.

***عبارت* countinue**

دستور **continue**، با اجرا شدن در عبارات **for, while** یا **do..while** از مابقی عبارات موجود در درون بدنه عبارت پرش کرده و کار را با حلقه بعد دنبال می‌کند. در عبارات **while** و **do..while**، شرط تکرار حلقه بلافاصله پس از اجرای **continue** ارزیابی می‌شود. در عبارات **for**، بخش افزایش دهنده، پس از ارزیابی شرط تکرار حلقه صورت می‌گیرد. این مورد تنها اختلاف مابین **for** و **while** است. قرار دادن اشتباه **continue** قبل از بخش افزایش در **while** می‌تواند سبب بوجود آمدن یک حلقه بی‌نهایت شود.

در برنامه ۱۴-۵ از دستور **continue** در یک عبارت **for** استفاده شده (خط 12) تا از عبارت خروجی خط 14 پرش شود، زمانیکه عبارت **if** در خطوط 12-11 تشخیص دهد که مقدار **count** برابر 5 شده است. زمانیکه عبارت **continue** اجرا شود، برنامه از مابقی بدنه **for** پرش خواهد کرد. کنترل برنامه با افزایش متغیر کنترلی عبارت **for** ادامه می‌یابد و بدنبال آن شرط تکرار حلقه صورت می‌گیرد تا تعیین کند آیا بایستی حلقه ادامه یابد یا خیر.

```
1  // Fig. 5.14: fig05_14.cpp
2  // continue statement terminating an iteration of a for statement.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     for ( int count = 1; count <= 10; count++ ) // loop 10 times
10    {
11       if ( count == 5 ) // if count is 5,
12          continue;      // skip remaining code in loop
13
14       cout << count << " ";
15    } // end for
16
17    cout << "\nUsed continue to skip printing 5" << endl;
18    return 0; // indicate successful termination
19 } // end main
```

```
1 2 3 4 6 7 8 9 10
Used continue to skip printing 5
```

شکل ۱۴-۵ | عبارت continue در یک عبارت for.

در بخش ۳-۵ مشخص کردیم که عبارت **while** می‌توانند در بسیاری از موارد بجای عبارت **for** بکار گرفته شود. یک استثناء زمانی رخ می‌دهد که عبارت افزایش‌دهنده در ساختار **while** بدنبال دستور **continue** آمده باشد. در اینحالت، عمل افزایش قبل از تست شرط ادامه حلقه اجرا نخواهد شد.

برنامه‌نویسی ایده‌ال

تعدادی از برنامه‌نویسان احساس می‌کنند که استفاده از ***break*** و ***continue*** برخلاف قواعد برنامه‌نویسی ساخت‌یافته است، چراکه می‌توان بجای بکارگیری این عبارات از تکنیک‌های برنامه‌نویسی ساخت‌یافته استفاده کرد به نوعی که دیگر نیازی به استفاده از آنها نباشد.

کارائی

*اگر عبارات **break** و **continue** درست بکار گرفته شوند، نسبت به تکنیک‌های ساخت‌یافته سریعتر اجرا می‌شوند.*

۵-۸ عملگرهای منطقی

تا بدین جا، فقط به معرفی *شرط‌های ساده‌ای*، همچون **count <= 10**، **total > 1000** و **number != sentinelValue** پرداخته‌ایم و هر عبارت انتخاب و تکرار فقط اقدام به ارزیابی یک شرط با یکی از عملگرهای >، <، =>، =< و != می‌کرد. برای تصمیم‌گیری که بر مبنای ارزیابی از چندین شرط بود، این بررسی‌ها را در عبارات جداگانه یا عبارتهای **if** یا **if…then** تودرتو انجام می‌دادیم.

به منظور رسیدگی موثرتر به شرط‌های پیچیده، C++ *عملگرهای منطقی* در نظر گرفته است. عملگرها عبارتند از **&&** (*AND* منطقی)، || (*OR* منطقی) و **!** (*NOT* منطقی، یا نفی منطقی). با طرح مثال‌های به بررسی عملکرد این عملگرها می‌پردازیم.

عملگر AND منطقی (&&)

فرض کنید می‌خواهیم از برقرار بودن دو شرط قبل از اینکه برنامه مسیر مشخصی را برای اجرا انتخاب کند، مطمئن شویم. در چنین حالتی، می‌توانیم از عملگر **&&** و بصورت زیر استفاده کنیم:

```
if ( gender == 1 && age >= 65 )
   seniorFemales++;
```

این عبارت **if** متشکل از دو شرط ساده است شرط **gender == 1** تعیین می‌کند که آیا شخص مونث است یا خیر و شرط **age >= 65** مشخص می‌کند که آیا شخص شهروند مسنی است یا خیر. ابتدا این دو شرط ساده مورد ارزیابی قرار می‌گیرند، چرا که تقدم عملگرهای == و =< به نسبت **&&** در مرتبه بالاتری قرار دارند. سپس عبارت **if** به بررسی ترکیبی شرط زیر می‌پردازد

```
gender == 1 && age >= 65
```

اگر فقط و فقط اگر هر دو شرط برقرار باشند، برقراری این شرط درست ارزیابی خواهد شد. هنگامی که ترکیب این شرط درست باشد، به مقدار **seniorFemales** یک واحد افزوده می‌شود. با این وجود، اگر فقط یکی از این دو شرط یا یکی از آنها برقرار نباشد (درست نباشد)، برنامه از انجام عمل افزایش صرفنظر کرده و به اجرای برنامه پس از عبارت **if** می‌پردازد. عبارت قبل را می‌توان با استفاده از پرانتزها بهتر کرد:

```
( gender == 1 ) && ( age >= 65 )
```

خطای برنامه‌نویسی

اگر چه 3<x<7 در جبر شرط درستی است، اما بدرستی در C++ ارزیابی نمی‌شود. برای ارزیابی صحیح در C++ باید از (3 < x && x < 7) استفاده کرد.

جدول شکل ۱۵-۵ عملکرد عملگر **&&** را نشان می‌دهد. در این جدول چهار حالت ممکنه از ترکیب مقادیر *false* و *true* بر روی *عبارت۱* و *عبارت۲* لیست شده‌اند. به این نوع جداول، جدول درستی گفته می‌شود.

عبارت۱	عبارت۲	عبارت۲ && عبارت۱
false	false	false
false	true	false
true	false	false
true	true	true

شکل ۱۵-۵| جدول درستی عملگر &&.

عملگر OR منطقی (||)

حال اجازه دهید تا به بررسی عملگر || (OR شرطی) بپردازیم. فرض کنید مایل هستیم تا قبل از انجام یک عمل مشخص از برقرار بودن هر دو شرط یا یکی از شرط‌ها (درست بودن) مطمئن شویم. در بخشی از برنامه زیر، از عملگر || استفاده شده است:

```
if ((semesterAverage >= 90) || (finalExam >= 90))
        cout << "Student grade is A" << endl;
```

این عبارت هم متشکل از دو شرط ساده است. با ارزیابی شرط **semesterAverage >= 90** مشخص می‌شود که آیا دانشجو به دلیل حفظ کارایی خود در طول ترم قادر به دریافت امتیاز "**A**" بوده است یا خیر. شرط **finalExam >= 90** تعیین می‌کند که آیا دانشجو در آزمون پایان ترم امتیاز "**A**" بدست آورده یا خیر. سپس عبارت **if** به بررسی ترکیبی شرط زیر می‌پردازد

```
(semesterAverage >= 90) || (finalExam >= 90)
```

و اگر یکی از شرط‌ها یا هر دو آنها برقرار باشند، نمره "**A**" به دانشجو اهداء می‌شود. دقت کنید که فقط در صورت برقرار نبودن هر دو شرط (*false* بودن)، عبارت **"Student grade is A"** چاپ نخواهد شد. جدول شکل ۱۶-۵، جدول درستی عملگر OR منطقی است.

عبارت۱	عبارت۲	عبارت۲ || عبارت۱
false	false	false
false	true	true
true	false	true
true	true	true

شکل ١٦-٥ | جدول درستی عملگر ||

عملگر **&&** از تقدم بالاتری نسبت به عملگر || برخوردار است. هر دو عملگر از چپ به راست ارزیابی می‌شوند. یک عبارت حاوی عملگرهای **&&** یا || فقط تا زمان شناخت درست بودن یا نبودن ارزیابی می‌گردد. از اینرو، در ارزیابی عبارت

```
(gender == 1) && (age >= 65)
```

اگر **gender** معادل با **"1"** نباشد (در اینصورت کل عبارت برقرار نخواهد بود)، ارزیابی عبارت دوم بیهود خواهد بود چرا که شرط عبارت اول برقرار نیست. ارزیابی عبارت یا شرط دوم فقط زمانی رخ می‌دهد که **gender** برابر **"1"** باشد (برای برقرار بودن کل عبارت هنوز هم باید شرط **age >= 65** برقرار گردد). این ویژگی در ارزیابی عبارات **&&** و **||** ارزیابی اتصالی نامیده می‌شود. در ضمن این ویژگی سبب افزایش کارائی می گردد.

کارائی

در عبارتی که از عملگر && استفاده می‌کند و شرط‌ها از هم متمایز هستند، آن شرطی را که احتمال برقرار نبودن آن بیشتر است در سمت چپ شرط قرار دهید. در عبارتی که از عملگر || استفاده می‌کند، شرطی را که احتمال برقرار بودن آن بیشتر است در سمت چپ شرط قرار دهید. در اینحالت از ارزیابی اتصالی استفاده شده و زمان اجرای برنامه کاهش می‌یابد.

عملگر نفی منطقی (!)

عملگر **!** (نفی منطقی) به برنامه‌نویس امکان می‌دهد تا نتیجه یک شرط را "معکوس" کند. برخلاف عملگرهای منطقی **&&** و **||** که از دو شرط استفاده می‌کنند (عملگرهای باینری هستند)، عملگر منطقی نفی یک عملگر غیرباینری است و فقط با یک عملوند بکار گرفته می‌شود. این عملگر قبل از یک شرط جای داده می‌شود. برای مثال به عبارت زیر دقت کنید:

```
if ( !( grade == sentinelValue ) )
  cout << "The next grade is " << grade << endl;
```

وجود پرانتزها در اطراف شرط **grade == sentinelValue** ضروری است، چراکه تقدم عملگر نفی از عملگر برابری (تساوی) بالاتر است. جدول شکل ۱۷-۵ جدول درستی عملگر نفی است.

عبارت	عبارت !
false	true
true	false

شکل ۱۷-۵ | جدول درستی عملگر نفی.

مثالی از عملگرهای منطقی

برنامه شکل ۱۸-۵ به توصیف عملگرهای منطقی مطرح شده در جداول درستی می‌پردازد. در خروجی هر عبارت ارزیابی شده و نتیجه بولی آن بنمایش در آمده است. بطور پیش فرض، مقادیر بولی true و false توسط **cout** و عملگر درج بصورت 1 و 0 بنمایش در آمده‌اند. در خط 11، از **boolalpha** کنترل کننده استریم استفاده کرده‌ایم. تا مشخص کنیم که مقدار هر عبارت بولی بایستی با کلمه "true" یا "false" بنمایش در آید. برای مثال، نتیجه عبارت **false && false** در خط 12 مقدار **false** است، از اینرو در دومین خط خروجی کلمه "false" بکار گرفته شده است. خطوط 15-11 جدول درستی **&&** را

تشکیل می‌دهند. خطوط 22-18 تولید کننده جدول درستی || هستند. خطوط 27-25 جدول درستی ! را ایجاد می کنند.

```
1  // Fig. 5.18: fig05_18.cpp
2  // Logical operators.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6  using std::boolalpha; // causes bool values to print as "true" or "false"
7
8  int main()
9  {
10    // create truth table for && (logical AND) operator
11    cout << boolalpha << "Logical AND (&&)"
12       << "\nfalse && false: " << ( false && false )
13       << "\nfalse && true: " << ( false && true )
14       << "\ntrue && false: " << ( true && false )
15       << "\ntrue && true: " << ( true && true ) << "\n\n";
16
17    // create truth table for || (logical OR) operator
18    cout << "Logical OR (||)"
19       << "\nfalse || false: " << ( false || false )
20       << "\nfalse || true: " << ( false || true )
21       << "\ntrue || false: " << ( true || false )
22       << "\ntrue || true: " << ( true || true ) << "\n\n";
23
24    // create truth table for ! (logical negation) operator
25    cout << "Logical NOT (!)"
26       << "\n!false: " << ( !false )
27       << "\n!true: " << ( !true ) << endl;
28    return 0; // indicate successful termination
29 } // end main
```

```
Logical AND (&&)
false && false: false
false && true:  false
true && false:  false
true && true:   true

Logical OR (||)
false || false: false
false || true:  true
true || false:  true
true || true:   true

Logical NOT (!)
!false: true
!true:  false
```

شکل ۱۸-۵ | عملگرهای منطقی.

تقدم و شرکت‌پذیری عملگرها

جدول به نمایش درآمده در شکل ۱۹-۵ تقدم عملگرهای معرفی شده تا بدین جا را نشان می‌دهد. تقدم عملگرها از بالا به پایین و به ترتیب کاهش می‌یابد.

عملگر	شرکت‌پذیری	نوع
()	left to right	parentheses
++ -- **static_cast**<*type*>**()**	right to left	unary postfix

++ -- + - !	right to left	unary prefix
* / %	left to right	multiplicative
+ -	left to right	additive
<< >>	left to right	insertion/extraction
< <= > >=	left to right	relational
== !=	left to right	equality
&&	left to right	logical AND
||	left to right	logical OR
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment
,	left to right	comma

شکل ۱۹-۵|تقدم و شرکت‌پذیری عملگرهای معرفی شده تا بدین فصل.

۹-۵ اشتباه گرفتن عملگر تساوی (==) و عملگر تخصیص (=)

یک نوع خطا وجود دارد که برنامه‌نویسان C++، بدون در نظر گرفتن میزان تجربه اکثراً با آن مواجه می‌شوند و بر این اساس یک بخش مجزا برای آن در نظر گرفته شده است. خطاهایی که بطور تصادفی با جابجا نوشتن عملگرهای == (تساوی) و = (تخصیص) رخ می‌دهند. اشتباهی که انجام آن موجب بروز خطای نحوی نمی‌شود و عباراتی که حاوی چنین اشتباهی هستند بدرستی کامپایلر می‌شوند و برنامه شروع بکار می‌کند، اما در زمان اجرا نتایج اشتباهی تولید می‌کنند و خطایی که با آن مواجه هستیم، خطای منطقی زمان اجرا است.

از دو نظر C++ به این مشکل رسیدگی می‌کند. یکی اینکه هر عبارتی که مقدار تولید می‌کند می‌تواند در بخش شرط هر عبارت کنترلی بکار گرفته شود. اگر مقدار عبارت، صفر باشد با آن همانند false و اگر مقدار عبارت، غیرصفر باشد با آن همان‌ند true رفتار می‌شود. دوم اینکه عبارت تخصیصی مقدار تولید می‌کند، یعنی مقداری به متغیر قرار گرفته در سمت چپ عملگر تخصیص، اختصاص می‌یابد. برای مثال، فرض کنید قصد نوشتن عبارت زیر را داشته باشیم

```
if ( payCode == 4 )
    cout<< "You get a bonus!" << endl;
```

اما تصادفاً بنویسیم

```
if ( paycode = 4 )
    cout << "You get a bonus!" << endl;
```

عبارت **if** اول بدرستی جایزه‌ای به شخصی که **paycode** آن برابر **4** است اهدا می‌کند. عبارت **if** دوم (که خطا دارد)، مبادرت به ارزیابی جمله تخصیص در شرط **if** با ثابت **4** می‌کند. هر مقداری غیر از صفر

بعنوان یک مقدار **true** تفسیر می‌شود، از اینرو شرط این عبارت **if** همیشه **true** بوده و همیشه این شخص جایزه دریافت می‌کند، صرفنظر از اینکه مقدار **payCode** آن چند باشد.

خطای برنامه‌نویسی

استفاده از عملگر == با هدف تخصیص و استفاده از عملگر = با هدف تساوی، خطای منطقی است.

اجتناب از خطا

معمولاً برنامه‌نویسان شرط‌هایی همانند 7 == x را با نام متغیر در سمت چپ و مقدار ثابت در سمت راست می‌نویسند. با برعکس نوشتن این ترتیب به نحوی که ثابت در سمت چپ و نام متغیر در سمت راست قرار گرفته باشد (7 == x) در صورتیکه برنامه‌نویس اشتباهاً مبادرت به نوشتن = بجای == کند، کامپایلر متوجه موضوع شده و با آن همانند یک خطای زمان کامپایل رفتار می‌کند، چرا که نمی‌توانید مقدار یک ثابت را تغییر دهید. با انجام اینکار می‌توان از رخ دادن خطاهای منطقی جلوگیری کرد.

به اسامی متغیرها، مقادیر سمت چپ (*lvalues*) گفته می‌شود چرا که از آنها در سمت چپ عملگر تخصیص استفاده می‌شود. به ثابت‌ها، مقادیر سمت راست (*rvalues*) گفته می‌شود، چرا که از آنها فقط در سمت راست عملگر تخصیص استفاده می‌شود. توجه کنید که lvalues را می‌توان به عنوان rvalues استفاده کرد، اما عکس این موضوع صادق نیست. فرض کنید برنامه‌نویس قصد دارد مقداری را به یک متغیر با یک عبارت ساده همانند عبارت زیر تخصیص دهد

```
x = 1;
```

اما بنویسد

```
x == 1;
```

در اینجا هم، خطای نحوی وجود ندارد. بجای آن، کامپایلر بسادگی مبادرت به ارزیابی عبارت رابطه‌ای می‌کند. اگر **x** معادل با 1 باشد، شرط برقرار بوده (true) و عبارت با مقدار true ارزیابی می‌شود. اگر **x** معادل 1 نباشد، شرط برقرار نبوده (false) و عبارت با مقدار **false** ارزیابی می‌شود. صرفنظر از مقدار عبارت، عملیات تخصیص صورت نمی‌گیرد و مقدار از دست می‌رود. مقدار **x** بدون تغییر باقی می‌ماند و می‌تواند در زمان اجرا، خطای منطقی تولید کند. متاسفانه روش مشخصی برای اجتناب از این مشکل وجود ندارد.

اجتناب از خطا

با استفاده از یک ویرایشگر متنی به بررسی تمام = های موجود در برنامه بپردازید و از استفاده صحیح آن مطمئن شوید.

۵-۱۰ چیکده برنامه‌نویسی ساخت‌یافته

همانند یک معمار که براساس دانش خود اقدام به طراحی ساختمان می‌کند، برنامه‌نویسان هم اقدام به طراحی و ایجاد برنامه‌ها می‌کنند و این در حالیست که دانش ما نسبت به معماری بسیار جوانتر بوده و از

ترکیب علوم مختلف ایجاد شده است.تا بدین جا آموختیم که برنامه‌نویسی ساخت‌یافته، برنامه‌های ایجاد می‌کند که درک، تست، خطایابی و اصلاح آنها به آسانی صورت می‌گیرد و اثبات این مسئله از طریق ریاضی هم ممکن است.

عبارتهای کنترلی ++C با استفاده از دیاگرام‌های فعالیت بصورت خلاصه در شکل ۲۰-۵ آورده شده‌اند. دایره‌های کوچکی که در تصاویر بکار رفته‌اند، نشان دهنده یک نقطه ورودی و یک نقطه خروجی برای هر عبارت هستند. اتصال جداگانه از نمادهای دیاگرام به صورت غیرقراردادی می‌تواند ما را به طرف برنامه‌های غیرساخت‌یافته سوق دهد. از اینرو یک برنامه‌نویس حرفه‌ای با انتخاب و ترکیب نمادها بفرمی که محدود به عبارتهای کنترل است و ایجاد برنامه‌های ساخت‌یافته به وسیله ترکیب عبارتهای کنترل در دو روش ساده می‌تواند به یک عبارت مناسب دست یابد.

برای سادگی کار، از یک نقطه ورودی و یک نقطه خروجی در عبارتهای کنترل استفاده می‌شود، از اینرو فقط یک راه برای ورود و یک راه برای خروج در هر عبارت کنترل وجود خواهد داشت. اتصال متوالی این عبارتهای کنترل و به شکل درآوردن برنامه‌های ساخت‌یافته آسان‌تر است، نقطه خروجی یک عبارت کنترل مستقیماً به نقطه ورودی یک عبارت کنترل دیگر متصل می‌شود. عبارتهای کنترل می‌توانند بصورت متوالی قرار گیرند (یکی پس از دیگری در یک برنامه) که به اینحالت *عبارت کنترل پشته‌ای* می‌گوئیم. قوانین برنامه‌نویسی ساخت‌یافته امکان کنترل عبارتهایی به نوع تودرتو یا آشیانه‌ای را فراهم می‌آورند.

شکل ۲۰-۵ | عبارتهای انتخاب و تکرار، تک ورودی/تک خروجی در ++C.

جدول شکل ۲۱-۵ حاوی قوانینی است که برای به شکل درآوردن صحیح برنامه‌های ساخت‌یافته ضروری هستند. در این قوانین فرض بر این است که نماد عمل برای ارائه هر نوع عمل اجرائی شامل ورودی/خروجی است.

قوانین شکل‌دهی برنامه‌های ساخت‌یافته
۱) شروع با ساده‌ترین دیاگرام فعالیت (شکل ۲۲-۵)
۲) هر نماد وضعیت عمل را می‌توان با دو نماد وضعیت عمل در حالت متوالی جایگزین کرد.
۳) هر وضعیت عمل را می‌توان جایگزین هر عبارت کنترلی کرد (توالی، عبارتهای **if...else**، **if**، **switch**، **while**، **do..while**، **for**)
۴) قوانین ۲ و ۳ را می‌توان هر چند بار که مایل هستیم و با هر ترتیبی تکرار کرد.

شکل ۲۱-۵ | قوانین برنامه‌نویسی ساخت‌یافته.

با بکار بردن قوانین ۲۱-۵ همیشه یک دیاگرام فعالیت مرتب و بفرم بلوکی ایجاد می‌شود. برای مثال نتیجه اعمال مکرر قانون دوم بر روی ساده‌ترین دیاگرام فعالیت (شکل ۲۲-۵)، یک دیاگرام فعالیت با تعدادی وضعیت عمل که بفرم متوالی و پشت سرهم قرار گرفته‌اند، است(شکل ۲۳-۵). دقت کنید که قانون دوم یک عبارت کنترلی پشته (بر روی هم قرار گرفته) ایجاد می‌کند، بنابراین به قانون دوم، *قانون پشته‌ای* گفته می‌شود.]

نکته:خطوط عمودی خط تیره در شکل ۲۳-۵ بخشی از UML نیستند. ما از آنها برای متمایز کردن چهار دیاگرام فعالیت استفاده کرده‌ایم که نشاندهنده قانون دوم از جدول ۲۱-۵ باشند.]

قانون سوم را قانون *تودرتو* یا *آشیانه‌ای* می‌نامند. بکار بردن قانون سوم به دفعات بر روی یک فلوچارت ساده، یک فلوچارت مرتب با عبارتهای کنترل تودرتو را نتیجه می‌دهد. برای مثال در شکل ۲۶-۵ مستطیل قرار گرفته در ساده‌ترین فلوچارت، در بار اول با یک عبارت دو انتخابی **(if/then)** جایگزین شده است. سپس مجدداً قانون سوم بر روی دو مستطیل موجود در عبارت دو انتخابی اعمال شده و هر کدام یک از این مستطیل‌ها با یک عبارت دو انتخابی جایگزین شده است. جعبه‌های خط چین که در اطراف هر عبارت دو انتخابی قرار گرفته‌اند نشان می‌دهند که بجای کدام مستطیل جایگزین شده‌اند.

شکل ۲۲-۵ | ساده‌ترین دیاگرام فعالیت.

شکل ۲۳-۵ | بکارگیری مکرر قانون دوم بر روی ساده‌ترین دیاگرام فعالیت.

شکل ۲۴-۵ نمایشی از انواع ساخت بلوکی به روش پشته است که از اعمال قانون دوم ایجاد شده و همچنین ساخت بلوکی به روش تودرتو با استفاده از قانون سوم را نشان می‌دهد. همچنین این شکل حاوی یک نوع از ساخت بلوکی روی هم قرار گرفته است که اینحالت نمی‌تواند در دیاگرام‌های فعالیت ساخت‌یافته بکار گرفته شود.

شکل ۲٤-۵ | اعمال مکرر قانون سوم بر روی ساده‌ترین دیاگرام فعالیت.

قانون چهارم یک عبارت تودرتوی بزرگتر، پیچیده‌تر و عمیق‌تر ایجاد می‌کند. دیاگرامی که با استفاده از قوانین جدول ۲۱-۵ ایجاد شود ترکیبی از تمام حالات ممکنه از دیاگرام‌های فعالیت خواهد بود و از اینرو تمام حالات برنامه‌های ساخت‌یافته را خواهد داشت. زیبائی این روش در این است که فقط با استفاده از هفت عبارت کنترلی ساده تک ورودی/تک خروجی ایجاد شده و اجازه ترکیب آنها در دو روش ساده را فراهم می‌آورد.

اگر قوانین جدول ۲۱-۵ بکار گرفته شوند، ایجاد یک دیاگرام فعالیات غیرساخت‌یافته (همانند شکل ۲۵-۵) غیرممکن خواهد بود. اگر مطمئن نیستید که یک دیاگرام ساخت‌یافته است، می‌توانید با اعمال قوانین جدول ۲۱-۵، بصورت معکوس و ساده کردن دیاگرام از این امر مطلع شوید. اگر دیاگرام قابلیت تبدیل به

یک دیاگرام فعالیت ساده را داشته باشد پس دیاگراتم اصلی ساخت‌یافته است و در غیر اینصورت ساخت‌یافته نمی‌باشد.

توسعه برنامه‌نویسی ساخت‌یافته به سادگی امکان‌پذیر است و Bohm و Jacopini نشان دادند که برای ایجاد برنامه‌ها فقط به سه شکل کنترلی نیاز است:

- توالی
- انتخاب
- تکرار

توالی حالت بدیهی دارد. انتخاب، با پیاده سازی یکی از سه روش زیر ممکن می‌شود:

- عبارت **if** (تک انتخابی)
- عبارت **if...else** (دو انتخابی)
- عبارت **switch** (چند انتخابی)

هر عبارتی که بتوان آنرا با **if...else** و **switch** نوشت، می‌توان با ترکیب عبارتهای **if** هم انجام داد (اگر چه شاید ظاهر خوبی نداشته باشد).

شکل ۲۵-۵ | دیاگرام فعالیت غیرساخت‌یافته.

تکرار می‌تواند با یکی از سه روش زیر پیاه سازی شود:

- عبارت **while**
- عبارت **do…while**
- عبارت **for**

هر عبارت تکراری را می‌توان با اعمال عبارت **while** پیاده سازی کرد (اگر چه شاید ظاهر خوبی نداشته باشد).

اگر بخواهیم از مطالب گفته شده نتیجه‌گیری نمائیم، می‌توان گفت کنترل‌های مورد نیاز در یک برنامه C++ می‌توانند موارد زیر باشند:

- توالی
- عبارت انتخاب **if**

- عبارت تکرار **while**

این عبارتهای کنترلی می‌توانند به دو روش پشته و تودرتو با یکدیگر بکار گرفته شوند که نشان از سر راست بودن و سادگی برنامه‌نویسی ساخت‌یافته دارد.

۵-۱۱ مبحث آموزشی مهندسی نرم‌افزار: شناسایی وضعیت و فعالیت شی‌ها در سیستم ATM

در بخش ۱۳-۴ مبادرت به شناسایی تعدادی از صفات کلاس مورد نیاز برای پیاده‌سازی سیستم ATM و افزودن آنها به دیاگرام کلاس در شکل ۲۴-۴ کردیم. در این بخش، نشان خواهیم داد که چگونه می‌توان این صفات را در وضعیت (حالت) یک شی عرضه کرد. همچنین مبادرت به شناسایی چندین وضعیت کلیدی خواهیم کرد که سبب اشتغال شی‌ها می‌شوند و در مورد تغییر وضعیت دادن شی‌ها در واکنش به انواع رویدادهای رخ داده در سیستم بحث خواهیم کرد. همچنین در ارتباط با روند کار، یا فعالیت‌ها صحبت می‌کنیم که شی‌ها در سیستم ATM انجام می‌دهند. فعالیت شی‌های **BalanceInquiry** و **Withdrawal** را در این بخش معرفی می‌کنیم، که از جمله فعالیت‌های کلیدی در سیستم ATM هستند (نمایش موجودی و برداشت پول).

دیاگرام‌های وضعیت ماشین

هر شی در یک سیستم در میان دنباله‌ای از وضعیت‌های مشخص شده حرکت می‌نماید. وضعیت جاری یک شی توسط مقادیر موجود در صفات شی در آن زمان مشخص می‌شود. *دیاگرام وضعیت ماشین* (که *دیاگرام وضعیت* نامیده می‌شود) مدل‌کننده وضعیت‌های کلیدی یک شی بوده و نشاندهنده شرایط محیطی است که شی در آن شرایط تغییر وضعیت می‌دهد. برخلاف دیاگرام‌های کلاس که بر روی ساختار اصلی سیستم تمرکز دارند، دیاگرام‌های وضعیت، مدل‌کننده برخی از رفتار سیستم هستند. شکل ۲۶-۵ یک دیاگرام وضعیت ساده است که برخی از وضعیت‌های یک شی از کلاس ATM را مدل کرده است. UML هر وضعیت را در یک دیاگرام وضعیت، بصورت یک *مستطیل گوشه‌گرد* با نام وضعیت جای گرفته در میان آن به نمایش در می‌آورد.

یک *دایره توپر* با یک فلش متصل شده نشاندهنده *وضعیت اولیه* می‌باشد. بخاطر دارید که این اطلاعات وضعیت را بصورت صفت **Boolean** برای **userAuthenticated** در دیاگرام کلاس شکل ۲۴-۴ مدل کرده‌ایم. این صفت با وضعیت *false* یا «کاربر تایید نشده است» بر طبق دیاگرام وضعیت مقداردهی اولیه می‌شود.

شکل ۲۶-۵ | دیاگرام وضعیت شی ATM.

جهت فلش‌ها نشاندهنده تراکنش‌های صورت گرفته مابین وضعیت‌ها هستند. یک شی می‌تواند از یک وضعیت در واکنش به رویدادهای مختلف که در سیستم رخ می‌دهند به وضعیت دیگر منتقل گردد. نام یا توصیف رویدادی که سبب تراکنش شده در کنار خطی که متناظر با تراکنش است نوشته می‌شود. برای مثال، شی ATM از وضعیت «کاربر تایید نشده» به وضعیت «کاربر تایید شده» پس از تایید کاربر از سوی پایگاه داده تغییر می‌یابد. از مستند نیازها بخاطر دارید که اعتبارسنجی کاربر از سوی پایگاه داده با مقایسه شماره حساب و PIN وارد شده از سوی کاربر با اطلاعات متناظر در پایگاه داده صورت می‌گیرد.

اگر پایگاه داده تشخیص دهد که کاربر به درستی شماره حساب و PIN را وارد کرده است، شی ATM به وضعیت «تایید کاربر» می‌رود و مقدار صفت **userAuthenticated** خود را به ***true*** تغییر می‌دهد. زمانیکه کاربر با انتخاب گزینه "**exit**" از منوی اصلی اقدام به خروج از سیستم می‌کند، شی ATM به وضعیت «کاربر تایید نشده است» باز می‌گردد و شرایط برای کاربر بعدی ATM مهیا می‌شود.

مهندسی نرم‌افزار

معمولاً طراحان نرم‌افزار هر حالت ممکنه را در دیاگرام‌های وضعیت قرار نمی‌دهند و فقط سعی در عرضه دیاگرام‌های وضعیت با اهمیت بیشتر یا وضعیت‌های پیچیده می‌کنند.

دیاگرام‌های فعالیت

همانند یک دیاگرام وضعیت، یک دیاگرام فعالی‌ت مبادرت به مدل کردن رفتار سیستم می‌کند. برخلاف دیاگرام وضعیت، دیاگرام فعالیت مبادرت به مدل‌سازی روند کار یک شی (توالی از رویدادها) در مدت زمان اجرای برنامه می‌کنند. دیاگرام فعالیت مدل‌کننده اعمال یک شی و ترتیب انجام کار توسط آن شی است. بخاطر دارید که از دیاگرام‌های فعالیت UML برای نمایش جریان کنترل عبارات کنترلی در فصل‌های ۴ و ۵ استفاده کردیم.

دیاگرام فعالیت به نمایش در آمده در شکل ۲۷-۵ فعالیت‌های انجام گرفته در مدت زمان اجرای یک تراکنش **BalanceInquiry** (نمایش موجودی) را مدل کرده است. فرض کرده‌ایم که یک شی **BalanceInquiry** در حال حاضر مقداردهی اولیه شده و یک شماره حساب معتبر به آن تخصیص یافته است، از اینرو شی می‌داند که کدام موجودی را بازیابی خواهد کرد. دیاگرام شامل اعمالی است که پس از انتخاب گزینه نمایش موجودی از سوی کاربر (از منوی اصلی) و قبل از اینکه ATM کاربر را به منوی اصلی بازگرداند، رخ می‌دهند. شی **BalanceInquiry** این اعمال را انجام نمی‌دهد، از اینرو ما آنها را در اینجا مدل نکرده‌ایم.

شکل ۲۷-۵ | دیاگرام فعالیت برای تراکنش BalanceInquiry.

دیاگرام با بازیابی موجودی در دسترس، حساب کاربر و از پایگاه داده آغاز می‌شود. سپس، **BalanceInquiry** کل موجودی را از حساب بازیابی می‌کند. در پایان تراکنش، میزان موجودی بر روی صفحه نمایش ظاهر می‌شود. با این عمل تراکنش کامل می‌شود.

UML یک عمل را در یک دیاگرام فعالیت بصورت وضعیت عمل شده با یک مستطیل که گوشه‌های چپ و راست آن حالت انحناء به خارج دارند نشان می‌دهد. هر وضعیت عمل حاوی یک جمله توضیح عمل است، برای مثال «دریافت میزان موجودی در حساب کاربر از پایگاه داده»، که مشخص می‌کند چه عملی انجام می‌شود. یک فلش، دو وضعیت عمل را به هم متصل کرده است و نشان‌دهنده ترتیب انجام اعمال است. دایره توپر (در بالای شکل ۲۷-۵) نشاندهنده وضعیت اولیه و آغاز روند کار می‌باشد. در این دیاگرام، ابتدا تراکنش مبادرت به اجرای عمل «دریافت میزان موجودی حساب کاربر از پایگاه داده» می‌کند. سپس تراکنش (مرحله دوم) مبادرت به بازیابی کل موجودی می‌نماید. در پایان تراکنش هر دو موجودی را بر روی صفحه نمایش ظاهر می‌سازد. دایره توپر احاطه شده در درون یک دایره (در پایین شکل ۲۷-۵) نشاندهنده وضعیت پایانی است، انتهای روند کار پس از اینکه شی، عمل‌های مدل شده را انجام داده است.

شکل ۲۸-۵ نمایشی از دیاگرام فعالیت برای تراکنش **Withdrawal** است (برداشت پول). فرض می‌کنیم که به شی **Withdrawal** در حال حاضر یک شماره حساب معتبر تخصیص یافته است. انتخاب گزینه برداشت پول از منوی اصلی یا برگشت دادن کاربر به منوی اصلی ATM را مدل نمی‌کنیم، چرا که این اعمال توسط شی **Wihtdrawal** صورت نمی‌گیرند. ابتدا تراکنش، منوی استاندارد میزان برداشت (شکل ۱۷-۲) و گزینه لغو تراکنش را به نمایش در می‌آورد. سپس تراکنش وارد منوی انتخابی از سوی کاربر می‌شود. اکنون جریان فعالیت به یک نماد تصمیم‌گیری می‌رسد. این نقطه تعیین‌کننده عمل بعدی بر مبنای محافظ شرط مربوطه است. اگر کاربر مبادرت به لغو تراکنش کند، سیستم پیغام مناسبی به نمایش در می‌آورد. سپس جریان لغو به یک نماد ادغام می‌رسد، مکانی که جریان فعالیت با سایر تراکنش‌های ممکنه پیوند می‌یابد. توجه کنید که یک نماد ادغام می‌تواند به هر تعداد فلش تراکنش ورودی داشته باشد، اما فقط فلش تراکنش از آن خارج می‌شود. نماد پایین دیاگرام تعیین می‌کند که آیا تراکنش باید از ابتدا تکرار شود یا خیر. زمانیکه کاربر تراکنش را لغو کند، شرط «پول پرداخت شده یا تراکنش لغو شده» برقرار گردیده، سپس تراکنش به وضعیت پایانی فعالیت می‌رسد.

اگر کاربر گزینه برداشت پول را از منو انتخاب کند، تراکنش مبادرت به تنظیم **amount** (صفتی از کلاس **Withdrawal** که قبلاً در شکل ۲۴-۴ مدل شده است) با میزان پول انتخابی از سوی کاربر می‌کند. سپس تراکنش، موجودی حساب کاربر را از پایگاه داده بدست می‌آورد (صفت **availableBalance** از شی **Account** کاربر). سپس جریان فعالیت به یک شرط دیگر می‌رسد. اگر میزان درخواستی کاربر از میزان موجودی بیشتر باشد، سیستم یک پیغام خطا در ارتباط با این موضوع به نمایش در می‌آورد. سپس کنترل با سایر جریان‌های فعالیت قبل از رسیدن به پایین‌ترین شرط موجود در دیاگرام ادغام می‌شود. شرط «پول پرداخت نشده و کاربر تراکنش را لغو نکرده است» برقرار می‌شود و از اینرو جریان فعالیت به بالای دیاگرام برگشت داده شده و تراکنش به کاربر اعلان می‌کند تا مقدار جدیدی وارد سازد.

اگر مقدار درخواستی کمتر یا برابر میزان موجودی کاربر باشد، تراکنش مبادرت به تست اینکه آیا پرداخت‌کننده اتوماتیک به میزان کافی پول نقد برای انجام تقاضای صورت گرفته دارد یا خیر انجام می‌دهد. اگر چنین نباشد، تراکنش یک پیغام خطای مناسب به نمایش در آورده و به نماد ادغام قبل از آخرین نماد تصمیم‌گیری منتقل می‌شود. چون پولی پرداخت نشده از اینرو جریان فعالیت به ابتدای دیاگرام فعالیت برگشت داده می‌شود و تراکنش به کاربر اطلاع می‌دهد تا مقدار جدیدی وارد سازد.

اگر پول به میزان کافی در اختیار باشد، تراکنش با پایگاه داده وارد تعامل شده و به میزان پول درخواستی، حساب کاربر را بدهکار می‌کند (از مقدار موجود در هر دو صفت **availableBalance** و **totalBalance** از شی **Account** کم می‌شود). سپس تراکنش مبادرت به پرداخت پول مورد تقاضا کرده و به کاربر فرمان می‌دهد تا پول را از دستگاه بردارد. سپس جریان اصلی با دو جریان خطا و جریان لغو ادغام می‌شود. در اینحالت، پول پرداخت شده است، از اینرو جریان فعالیت به وضعیت پایانی رسیده است.

شکل ۲۸-۵ | دیاگرام فعالیت برای تراکنش Withdrawal.

اولین گام‌ها را برای مدل کردن رفتار سیستم ATM و نحوه نمایش صفات در ضمن فعالیت را برداشته‌ایم. در بخش ۲۲-۶ به عملیات کلاس‌ها رسیدگی می‌کنیم تا مدل کاملتری از رفتار سیستم ایجاد نمائیم.

تمرینات خودآزمایی مبحث مهندسی نرم‌افزار

۵-۱ تعیین کنید آیا عبارت زیر صحیح است یا اشتباه. در صورت اشتباه بودن علت را توضیح دهید: دیاگرام وضعیت مدل‌کننده جنبه‌های ساختاری یک سیستم است.

۵-۲ دیاگرام فعالیت مدل‌کننده ———— است که یک شی انجام می‌دهد و ترتیب انجام آن را مشخص می‌کند.

a) اعمال

b) صفات

c) وضعیت

d) وضعیت تراکنش

۵-۳ براساس مستند نیازها، یک دیاگرام فعالیت برای تراکنش سپرده‌گذاری ایجاد کنید.

پاسخ خودآزمایی مبحث آموزشی مهندسی نرم‌افزار

۵-۱ اشتباه. دیاگرام‌های وضعیت مدل‌کننده برخی از رفتار سیستم هستند.

۵-۲ a

۵-۳ شکل ۲۹-۵ دیاگرام فعالیت برای تراکنش سپرده‌گذاری است. دیاگرام، مدل‌کننده اعمالی است که پس از انتخاب گزینه سپرده‌گذاری از منوی اصلی و قبل از بازگرداندن کاربر به منوی اصلی توسط ATM است.

شکل ۲۹-۵| دیاگرام فعالیت برای تراکنش سپرده‌گذاری (Deposit).

خودآزمایی

۵-۱ کدامیک از عبارات زیر صحیح و کدامیک اشتباه است. اگر عبارتی اشتباه است علت آنرا توضیح دهید.

a) عبارت **default** باید در عبارت انتخاب **switch** بکار گرفته شود.

b)عبارت **break** در حالت **default** ساختار **switch** برای خروج صحیح از **switch** ضروری است.

c) عبارت **x > y && a < b** برقرار است که خواه **x > y** برقرار باشد یا نباشد یا اینکه **a < b** برقرار باشد.

d) عبارتی که حاوی عملگر || است، در صورتیکه یکی از عملوندها یا هر دو عملوند برقرار (صحیح) باشند، کلاً صحیح ارزیابی می‌شود.

۵-۲ عبارت یا عباراتی در C++ بنویسید که موارد خواسته شده زیر را برآورده سازند:

a) محاسبه مجموع اعداد فرد از 99 تا 1 با استفاده از یک عبارت **for**. با فرض اینکه متغیرهای **sum** و **count** از نوع صحیح اعلان شده‌اند.

b) چاپ مقدار 333.546372 در میدانی به پهنای 15 کارکتر با دقت 2 ,1 و 3. هر عدد را در یک خط چاپ کنید.هر عدد را در میدان خود از چپ تراز کنید.

c) مقدار 2.5 را بتوان 3 برسانید. با استفاده از تابع **pow**. نتیجه را با دقت 2 در میدانی به پهنای 10 چاپ کنید.

d) چاپ اعداد از 20 تا 1 با استفاده از یک حلقه **while** و متغیر شمارنده **x**. با فرض اینکه متغیر **x** اعلان شده اما مقداردهی اولیه نشده است. هر پنج عدد در یک سطر چاپ شود.

e) تمرین c را با استفاده از عبارت **for** تکرار کنید.

۵-۳ خطا یا خطاهای موجود در کدهای زیر را یافته و آنها را اصلاح کنید.

a)

```
x=1;
while(x,=10);
  x++;
}
b)
for(y=1; y!=1.0;y+=.1)
   count<<y<<endl;
c)
switch(n)
{
 case1:
     cout<< "The number is 1"<<endl;
 case2:
     cout<<"The number is 2"<<endl;
     break;
 default:
     cout<<"The number is not 1 or 2"<<endl;
     break;
}
```

d) کد زیر باید مقادیر ۱ تا ۱۰ را چاپ کند

```
n=1;
while(n<10)
 cout<<n++<<endl;
```

پاسخ خودآزمایی

۵-۱ a) اشتباه. استفاده از **default** اختیاری است. b) اشتباه. عبارت break برای خروج از ساختار switch بکار گرفته می‌شود. c) اشتباه. به هنگام استفاده از عملگر && باید هر دو عبارت رابطه‌ای برای برقرار بودن کل عبارت، برقرار باشند. d) صحیح.

۵-۲

```
a)
sum=0;
for(count=1;count<=99;count+=2)
   sum+=count;
b)
cout<<fixed<<left
<<setprecision(1)<<setw(15)<<333.5467372
<< setprecision(2)<<setw(15)<<333.5467372
<< setprecision(3)<<setw(15)<<333.5467372
<<endl;
```

خروجی

```
333.5      333.55      333.546
```

```
c)
cout<<fixed<<setprecision(2)
      <<setw(10)<<pow(2.5,3)
      <<endl;
```

خروجی

```
15.63
```

```
d)
x=1;
while(x<=20)
{
 cout<<x;
 if(x%5==0)
   cout<<endl;
 else
    cout<<'\t';
 x++;
}
e)
for(x=1;x<=20;x++)
{
 cout<<x;
 if(x%5==0)
  cout<<endl;
 else
  cout<<'\t';
}
```

یا

```
 for(x=1;x<=20;x++)
{
 if(x%5==0)
  cout<<x<<endl;
 else
  cout<<x<<'\t';
}
```

۵-۳

(a

خطا: سیمکولن پس از سرآیند while، حلقه بی‌نهایت بوجود می‌آورد.

ااصلاح: جایگزین کردن سیمکولن با یک } یا حذف ; و {

(b

خطا: استفاده از یک عدد اعشاری در کنترل عبارت تکرار for

اصلاح: استفاده از یک عدد صحیح و انجام محاسبه مناسب به ترتیبی که مقادیر مورد نظر بدست آیند.

```
For(y=1; y!=10;y++)
 cout<<(static_cast<double>(y)/10)<<endl;
```

(c

خطا: فاقد عبارت break در اولین case.

ااصلاح: افزودن یک break در انتهای عبارت اولین case. توجه کنید در صورتیکه برنامه‌نویس مایل بوده باشد که عبارت :case 2 همیشه پس از اجرای :case 1 اجرا شود، اینکار خطا محسوب نمی‌شود.

(d

خطا: از عملگر رابطه‌ای صحیحی استفاده نشده است (در شرط تکرار حلقه while).

ااصلاح: استفاده از => بجای > یا تغییر 10 به 11.

تمرینات

۵-۴ خطا یا خطاهای موجود در عبارات زیر را پیدا کنید:

a)
```
for (x=100, x>=1,x++)
      cout << x << endi;
```

b) کد زیر باید مقادیر صحیح زوج یا فرد را چاپ کند

```
switch (value %2)
{
case 0:
    cout<< "Even integer" << endl;
case 1:
    cout << " Odd integer" << endl;
}
```

c) کد زیر باید مقادیر فرد ۱۹ تا ۱ را چاپ کند

```
for (x=19; x>=1; x+=2)
    cout << x << endl;
```

d) کد زیر باید مقادیر زوج ۲ تا ۱۰۰ را چاپ کند.

```
courter= 2;
do
{
cout << counter << endl;
counter +=2;
} while (counter < 100);
```

۵-۵ برنامه‌ای بنویسید که با استفاده از یک عبارت for مجموع، توالی از مقادیر صحیح را محاسبه کند. فرض کنید که نخستین عدد، نشاندهنده تعداد مقادیری باشد که وارد خواهد شد. برنامه شما باید در هر عبارت ورودی یک مقدار دریافت کند. برای مثال، نمونه می‌تواند بصورت زیر باشد

```
5   100   200   300   400  500
```

در این دنباله، 5 نشان می‌دهد که پنج مقدار وارد و جمع خواهند شد.

۵-۶ برنامه‌ای بنویسید که با استفاده از یک عبارت **for** مبادرت به محاسبه و چاپ میانگین چندین مقدار صحیح کند. فرض کنید مقدار مراقبتی 9999 باشد. یک ورودی نمونه می‌تواند بصورت زیر باشد.

```
10   8   11   7   9   9999
```

۵-۷ برنامه زیر چه کاری انجام می‌دهد؟

```
// Exercise 5.7: ex05_07.cpp
// What does this program print?
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

int main()
{
   int x; // declare x
   int y; // declare y

   // prompt user for input
   cout << "Enter two integers in the range 1-20: ";
```

```
   cin >> x >> y; // read values for x and y

   for ( int i = 1; i <= y; i++ ) // count from 1 to y
   {
      for ( int j = 1; j <= x; j++ ) // count from 1 to x
         cout << '@'; // output @

      cout << endl; // begin new line
   } // end outer for

   return 0; // indicate successful termination
} // end main
```

۵-۸ برنامه‌ای بنویسید که با استفاده از یک عبارت for کوچکترین مقدار را از میان چندین مقدار صحیح پیدا کند. فرض کنید که اولین مقدار، نشاندهنده تعداد مقادیر باقیمانده باشد و این مقدار در مقایسه شرکت نمی‌کند.

۵-۹ برنامه‌ای بنویسید که با استفاده از یک عبارت for مبادرت به محاسبه و چاپ حاصلضرب اعداد فرد قرار گرفته از 1 تا 15 نماید.

۵-۱۰ برنامه‌ای بنویسید که مقدار فاکتوریل اعداد از 5 تا 1 را محاسبه کنید. خروجی برنامه را در فرمت جدولی به نمایش درآورید.

۵-۱۱ برنامه محاسبه سود مطرح شده در بخش ۴-۵ را به نحوی تغییر دهید که محاسبه سود را برای درصدهای 9 ,8 ,7 ,6 ,5 و 10 بدست آورد. از یک عبارت for برای ایجاد تغییر در سود استفاده کنید.

۵-۱۲ برنامه‌ای بنویسید که الگوهای زیر را بصورت جداگانه و زیر هم به نمایش درآورد. از حلقه‌های **for** برای ایجاد الگوها استفاده کنید. برنامه را به نحوی تغییر دهید تا هر چهار الگو را با یکدیگر و در کنار هم چاپ کند.

```
(A)         (B)         (C)         (D)

*           **********  **********          *
**          *********   *********          **
***         ********    ********          ***
****        *******     *******          ****
*****       ******      ******          *****
******      *****       *****          ******
*******     ****        ****          *******
********    ***         ***          ********
*********   **          **          *********
**********  *           *          **********
```

۵-۱۳ یکی از برنامه‌های جالب کامپیوتری ترسیم گراف‌ها و نمودارها است. برنامه‌ای بنویسید که پنج عدد دریافت کند (هر یک مابین 1 و 30). فرض کنید که کاربر فقط مقادیر معتبر وارد می‌سازد. برای هر عددی که خوانده می‌شود، بایستی برنامه یک خط حاوی کاراکتر ستاره برابر با آن عدد چاپ کند. برای مثال اگر برنامه عدد 7 را دریافت کند، باید ******** را چاپ نماید.

۵-۱۴ یک فروشگاه که توسط سیستم پستی تجارت می‌کند پنج محصول مختلف عرضه می‌کند که قیمت خرده‌فروشی آنها به ترتیب زیر است:
محصول 1 به قیمت 2.98$، محصول 2 به قیمت 4.50$، محصول 3 به قیمت 9.98$، محصول 4 به قیمت 4.49$ و محصول 5 به قیمت 6.87$. برنامه‌ای بنویسید که جهت مقادیر را بصورت زیر دریافت کند:

a) شماره محصول

b) تعداد فروخته شده

در این برنامه باید از یک عبارت **switch** برای تعیین قیمت خرده‌فروشی برای هر محصول استفاده کنید. برنامه باید مبلغ کل فروش تمام محصولات را محاسبه و به نمایش در آورد. از یک حلقه کنترل مراقبتی برای تعیین زمانیکه باید حلقه متوقف شده و نتایج نهایی به نمایش در آیند، استفاده کنید.

۵-۱۵ برنامه GradeBook شکل‌های ۵-۹ الی ۵-۱۱ را به نحوی تغییر دهید که میانگین را برای مجموعه امتیازات محاسبه نماید. امتیاز A دارای ارزش 4، امتیاز B دارای ارزش 3 و الی آخر است.

۵-۱۶ برنامه شکل ۵-۶ را به نحوی اصلاح کنید که فقط از مقادیر صحیح برای محاسبه نرخ سود استفاده کند.

۵-۱۷ با فرض i=1، j=2، k=3 و m=2 باشد. هر کدامیک از عبارت زیر چه چیزی چاپ خواهند کرد؟ آیا وجود پرانتزها ضروری است؟

a) `cout << (i==1) << endl;`
b) `cout << (i==3) << endl;`
c) `cout << (i>=1 && j<4) << endl;`
d) `cout << (m<=99 && k<m) << endl;`
e) `cout << (j>=i || k==m) << endl;`
f) `cout << (k+m<j || 3-j>=k) << endl;`
g) `cout << (!m) <<endl;`
h) `cout << (!(j-m)) << endl;`
i) `cout << (!(K>m)) << endl;`

۵-۱۸ برنامه‌ای بنویسید که یک جدول از اعداد باینری، اکتال و هگزا دسیمال معادل با اعداد دسیمال در محدوده 1 الی 256 را به نمایش درآورد.

۵-۱۹ مقدار π را از دنباله بی‌نهایت زیر محاسبه کنید:

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{11} + \ldots$$

مقدار π را پس از 1000 عبارت اول در این دنباله، چاپ کنید.

۵-۲۰ (سه‌گانه فیثاغورث) یک مثلث راست گوشه می‌تواند اضلاعی داشته باشد که همگی از نوع صحیح باشند. به این سه نوع ضلع سه‌گانه فیثاغورث می‌گویند. این سه ضلع باید رابطه‌ای را برآورد سازند که در آن مجموع مربع دو ضلع برابر با مربع وتر باشد. تمام مثلث‌های راست گوشه برای side2, side1 و hypotenuse (وتر) را که بزرگتر از 500 نمی‌باشند را پیدا کنید. از یک **for** تودرتو استفاده نمائید.

۵-۲۱ در شرکتی پرداختی‌ها برای مدیران (کسانی که حقوق هفتگی ثابت دریافت می‌کند)، کارگران ساعتی (کسانی که یک دستمزد ثابت ساعتی برای کار تا سقف 40 ساعت دریافت کرده و برای هر ساعت اضافه کاری 1.5 برابر دستمزد ساعتی دریافتی دارند)، کارگران کمیسیون (کسانی که $250 به همراه 5.7 درصد از حقوق ناخالص هفتگی دریافت می‌کنند) یا مقاطعه‌کارها (کسانی که یک مبلغ ثابت برای هر ایتم تولیدی دریافت می‌کنند، هر مقاطعه کار در این شرکت فقط بر روی یک نوع ایتم کار می‌کند) صورت می‌گیرد. برنامه‌ای بنویسید که حقوق هفتگی هر کارمند را محاسبه کند. در ابتدای کار از تعداد کارمندان اطلاعی ندارید. هر کارمندی دارای کد پرداختی متعلق به خود است. مدیران دارای کد 1، کارگران ساعتی دارای کد 2، کارگران کمیسیون دارای کد 3 و مقاطعه‌کاران دارای کد 4 هستند. با استفاده از یک عبارت **switch** مبادرت به محاسبه حقوق هر کارمند براساس کد کنید. در درون **switch** به کاربر اعلان کنید تا اطلاعات مورد نیاز برای محاسبه حقوق کارمند را وارد سازد.

۵-۲۲ (قوانین دمرگان) در این فصل در مورد عملگرهای منطقی **&&**، **||** و **!** صحبت کردیم. گاهی اوقات استفاده از قوانین دمرگان می‌تواند نحوه استفاده از عبارات منطقی را مناسبتر سازد. این قوانین نشان می‌دهند که عبارت **(conditional1 && conditional2)!** بطور منطقی برابر است با عبارت **(!conditional1||!conditional2)**. همچنین عبارت **(conditional1||conditional2)!** بطور منطقی برابر است با عبارت **(!conditional1 && !conditional2)**. با استفاده از قوانین دمرگان عبارات معادل برای هر یک از موارد زیر بنویسید، سپس برنامه‌ای بنویسید تا نشان دهد که عبارت اصلی و عبارت جدید در هر مورد با هم برابر هستند.

a) `!( x < 5 ) && !( y >= 7)`
b) `!( a == b ) || !( g != 5 )`
c) `!( ( x <= 8 ) && ( y> 4 ) )`
d) `!( ( i > 4 ) || ( j <= 6 ) )`

۵-۲۳ برنامه‌ای بنویسید که لوزی شکل زیر را چاپ کند.

```
    *
   ***
  *****
 *******
*********
 *******
  *****
   ***
    *
```

۵-۲۴ برنامه نوشته شده در تمرین ۵-۲۳ را به نحوی تغییر دهید تا یک عدد فرد در محدودهٔ 1 تا 19 که مشخص کننده تعداد سطرها در لوزی است دریافت کرده، سپس لوزی را با آن سایز به نمایش در آورد.

۵-۲۵ انتقادی که از عبارات break و continue می‌شود این است که آنها را غیرساختیافته می‌دانند. در واقع می‌توان همیشه این عبارات را با عبارات ساختیافته جایگزین کرد، اگر چه شاید انجام چنین کاری چندان استادانه نباشد. توضیح دهید که چگونه می‌توانید هر عبارت break را از حلقه‌ای در یک برنامه حذف کرده و آن را با یک عبارت معادل ساختیافته جایگزین کنید.

۵-۲۶ این بخش از کد چه کاری انجام می‌دهد؟

```
for ( int i = 1; i <= 5; i++ )
{
   for ( int j = 1; j <= 3; j++ )
   {
      for ( int j = 1; k <= 4; j++ )
         cout << '*';

      cout << endl ;
   } // end inner for

   cout << endl;
} // end outer for
```

۵-۲۷ توضیح دهید که به چه روشی می‌توان هر عبارت **continue** را از یک حلقه در برنامه حذف کرده و آن را با یک عبارت ساختیافته جایگزین کرد.

۵-۲۸ برنامه‌ای بنویسید که از عبارات تکرار و switch برای چاپ آهنگ The Twelve Days of Chrismas استفاده کند. باید از یک عبارت switch برای چاپ روز (یعنی "First" , "Second" ,...) استفاده شود. باید از یک عبارت switch جداگانه برای چاپ مابقی شعر استفاده کنید. برای داشتن شعر کامل این آهنگ می‌توانید به وب‌سایت www.12days.com/ library/carols/12dayssofxmas مراجعه کنید.

۵-۲۹ مطابق یک افسانه، Peter Minuit در سال 1626 جزیره Manhattan را به قیمت 24 دلار خریداری کرده است. آیا وی سرمایه‌گذاری خوبی انجام داده است؟ برای پاسخ دادن به این سوال برنامه محاسبه سود مطرح شده در شکل ۶-۵ را برای شروع کار با سرمایه اصلی $24.00 تنظیم کرده و مبلغ سود سپرده‌گذاری را تا این سال محاسبه کنید.

فصل ششم

توابع و مکانیزم بازگشتی

اهداف

- ایجاد مدولار برنامه‌ها با بخش‌هایی بنام توابع.
- استفاده از توابع ریاضی موجود در کتابخانه استاندارد C++.
- ایجاد توابع با پارامترهای مضاعف.
- آشنایی با مکانیزم‌های ارسال اطلاعات مابین توابع و برگشت نتایج.
- آشنایی با مکانیزم فراخوانی و برگشت توابع.
- تکنیک‌های شبیه‌سازی بکار رفته در ایجاد اعداد تصادفی.
- آشنایی با مبحث قلمرو.
- آشنایی با نحوه عملکرد و نوشتن توابعی بازگشتی.

رئوس مطالب

٦-١ مقدمه

بیشتر برنامه‌های کامپیوتری که مسائل دنیای واقعی را برطرف می‌کنند به نسبت برنامه‌هایی که در چند فصل آغازین ارائه شدند، بسیار بزرگتر و پیچیده‌تر هستند. تجربه نشان داده بهترین روش برای ساخت و نگهداری یک برنامه بزرگ این است که برنامه به قسمت‌های کوچکتر تقسیم شود به نحوی که هر قسمت وظیفه خاصی داشته باشد. در اینحالت می‌توان بطرز شایسته‌ای بر برنامه مدیریت داشت. این روش به نام

روش تقسیم و غلبه (divide and conquer) معروف است. در این فصل با روش‌هایی آشنا خواهید شد که در آن طراحی، تکمیل، اجرا و نگهداری برنامه‌های بزرگ به آسانی صورت می‌گیرد.

به بررسی بخشی از توابع ریاضی کتابخانه استاندارد C++،که برخی از آنها به بیش از یک پارامتر نیاز دارند، می‌پردازیم. سپس با نحوه اعلان یک تابع با بیش از یک پارامتر آشنا خواهید شد. همچنین اطلاعات بیشتری در مورد نمونه‌های اولیه تابع بدست آورده و به بررسی نحوه عملکرد کامپایلر در تبدیل نوع آرگومان تابع فراخوانی شده به نوع پارامترهای تابع می‌پردازیم.

سپس، به بررسی مختصر تکنیک‌های شبیه‌سازی با اعداد تصادفی پرداخته و نسخه‌ای از بازی پرتاب تاس بنام craps را ایجاد می‌کنیم. در این برنامه از اغلب تکنیک‌های برنامه‌نویسی که تا بدین‌جا آموخته‌اید استفاده شده است.

در ادامه، به معرفی کلاس‌ها و قوانین قلمرو در C++ می‌پردازیم. قانون قلمرو، تعیین کننده، مدت زمانی است که در طی آن یک شی در حافظه وجود دارد و نیز شناسه‌ای است که در برنامه می‌تواند مورد مراجعه قرار گیرد. همچنین خواهید آموخت که چگونه C++ می‌تواند تابع در حال اجرا را ردگیری کند، نحوه نگهداری پارامترها و سایر متغیرهای محلی در درون حافظه و مدیریت آنها را چگونه انجام می‌دهد، و چگونه یک تابع پس از کامل شدن اجرا می‌داند که به چه مکانی باید بازگردد. در ادامه به بررسی دو مبحثی که به بهبود کارایی برنامه کمک می‌کنند، خواهیم پرداخت، توابع inline که می‌توانند سربارگذاری فراخوانی تابع را حذف سازند و پارامترهای مراجعه که می‌توانند برای ارسال ایتم‌های بزرگ داده‌ی به توابع مورد استفاده قرار گیرند تا کارائی تابع افزایش یابد.

امکان دارد در تعدادی از برنامه‌ها از چند تابع همنام استفاده کنید. این تکنیک، سربارگذاری تابع نامیده می‌شود، و از طرف برنامه‌نویسان برای پیاده‌سازی توابعی که وظایف مشابهی را با آرگومان‌هایی از نوع‌های متفاوت یا با تعداد متفاوتی از آرگومان‌ها انجام می‌دهند بکار گرفته می‌شود. به بررسی الگوهای تابع هم خواهیم پرداخت. الگوی تابع، مکانیزمی برای تعریف خانواده‌ای از توابع سربارگذاری شده است. در بخش پایانی این فصل به بررسی توابعی که خود را فراخوانی می‌کنند، چه بصورت مستقیم یا غیرمستقیم (از طریق تابع دیگر) پرداخته شده است، مبحثی که بعنوان بازگشتی شناخته می‌شود و بطور کاملتر در دوره‌های بالاتر علوم کامپیوتر توضیح داده می‌شود.

۶-۲ کامپونت‌های برنامه در C++

برنامه‌های C++ متشکل از قسمت‌های متعددی از جمله توابع و کلاس‌ها هستند. برنامه‌نویس اقدام به ایجاد و ترکیب توابع و کلاس‌های جدید با کلاس‌های از قبل آماده شده موجود در کتابخانه استاندارد C++ می‌کند. در این فصل، تمرکز ما بر روی توابع است.

کتابخانه استاندارد C++ حاوی کلکسیون با ارزشی از توابع به منظور انجام محاسبات ریاضی، کار با رشته‌ها، کار با کاراکترها، عملیات ورودی/خروجی، تست و بررسی خطا و بسیاری از کاربردهای مناسب دیگر است. این کتابخانه بدلیل تدارک دیدن نیازهای متعدد یک برنامه‌نویس، کار برنامه‌نویس را آسانتر می‌کند. توابع کتابخانه استاندارد C++ بعنوان بخشی از محیط برنامه‌نویسی C++ تدارک دیده شده‌اند.

مهندسی نرم‌افزار

سعی کنید با کلکسیون با ارزش کلاس‌ها و توابع موجود در کتابخانه استاندارد C++ حتماً آشنا شوید.

مهندسی نرم‌افزار

برای ارتقای قابلیت استفاده مجدد از نرم افزار، هر تابع باید مختص انجام یک کار در نظر گرفته شده باشد، وظیفه آن بخوبی تعریف شده باشد، و نام تابع باید بطور موثر بیان‌کننده وظیفه تابع باشد. چنین توابعی کار نوشتن، تست، دیباگ و نگهداری برنامه ها را آسانتر می‌کنند.

اجتناب از خطا

تست و خطایابی یک تابع کوچک که یک وظیفه را انجام می‌دهد به نسبت یک تابع بزرگتر که وظایف متعددی دارد، راحتر است.

مهندسی نرم‌افزار

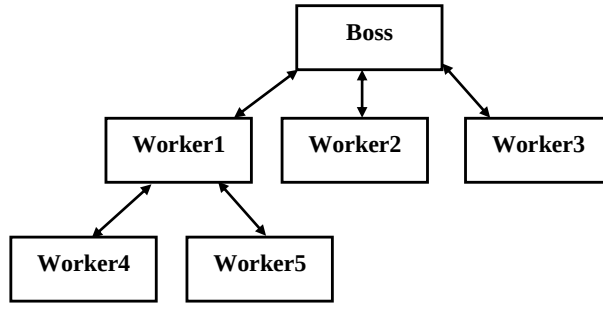
اگر نمی‌توانید نام دقیق و مناسبی برای کاری که تابع انجام می‌دهد انتخاب کنید، احتمالا تابع بیش از یک وظیفه بر عهده دارد. بهتر است چنین توابعی را به توابع کوچکتر تبدیل کرد.

اگر چه کتابخانه استاندار C++ توابع متعددی در نظر گرفته که وظایف زیادی به انجام می‌رسانند، اما نمی‌تواند هر آنچه را که یک برنامه‌نویس به آن نیاز دارد، در اختیار وی قرار دهد، از اینرو به برنامه‌نویسان امکان داده شده تا توابع متعلق به خود را ایجاد کنند تا نیازهای آنها را در حل مسائل خاص برطرف سازند. به این نوع از توابع، *توابع تعریف شده توسط کاربر* یا *توابع تعریف شده توسط برنامه‌نویس* گفته می‌شود. برنامه‌نویسان با نوشتن توابع قصد تعریف وظایف مشخص در یک برنامه را دارند و ممکن است از آنها در طول اجرای برنامه چندین بار استفاده کنند. اگر چه یک تابع ممکن است در چندین نقطه برنامه به دفعات بکار گرفته شود، اما عبارات تابع فقط یک بار نوشته می‌شوند.

به چند دلیل برای مدولار کردن یک برنامه از توابع استفاده می‌شود. در روش تقسیم و غلبه مدیریت توسعه برنامه بسیار بهتر صورت می‌گیرد. دلیل دیگر *استفاده مجدد از نرم‌افزار* است (استفاده دوباره از تابع موجود در ایجاد بلوک‌های یک برنامه جدید). در صورتیکه از تابع مناسب و قابل اعتماد بجای نوشتن کد متعلق بخود استفاده شود، یک برنامه با توابعی استاندارد بدست می‌آید. برای مثال، مجبور نیستیم تا نحوه خواندن یک رشته

متنی از صفحه‌کلید را تعریف کنیم، چراکه C++ دارای تابعی بنام **getline** در فایل سرآیند <**string**> به همین منظور است. سومین دلیل در استفاده از تابع، اجتناب از نوشتن کد تکراری در برنامه است. کدی که بعنوان یک تابع نوشته می‌شود، از این قابلیت برخوردار است که از مکان‌های مختلف برنامه فراخوانی شده و اجرا گردد.

یک تابع با فراخوانی فعال می‌شود و وظیفه‌ای که به آن منظور طراحی شده است به اجرا در می‌آورد. فراخوانی یک تابع مستلزم تهیه نام تابع و اطلاعات مورد نیاز تابع (آرگومان‌ها) از سوی فراخواننده تابع است. هنگامی که تابع وظیفه خود را به اتمام رساند، کنترل را به فراخواننده باز می‌گرداند (فراخواننده تابع). گاهی اوقات، تابع می‌تواند نتیجه‌ای به فراخواننده خود نیز برگشت دهد. سلسله مراتب مدیریت شباهت زیادی به روش عملکرد توابع دارد. بدین ترتیب که کارفرما (Boss) که نقش فراخواننده را دارد، از یک کارگر (worker) که نقش فراخوانده شده را دارد، می‌خواهد کاری انجام داده و نتیجه را پس از پایان کار گزارش دهد. کارفرما اطلاعی از اینکه کدام کارگر عمل درخواست شده را انجام می‌دهد ندارد، چرا که ممکن است کارگر فراخوانده شده، کارگرهای دیگری را فرا بخواند در حالیکه کارفرما از این مسائل اطلاعی ندارد. بزودی، نشان خواهیم داد که چگونه این روش *پنهان کردن جزئیات پیاده سازی* نقش مناسبی در مهندسی نرم‌افزار ایفا می‌کند. در شکل ۱-۶ رابطه تابع **Boss** با توابعی کارگر **Worker1**، **Worker2** و **Worker3** در روش سلسله مراتب دیده می‌شود. دقت کنید که **Worker1** نقش یک تابع کارفرما را برای توابعی **Worker4** و **Worker5** بازی می‌کند.



شکل ۱-٦ | سلسله مراتب رابطه تابع کارفرما/تابع کارگر.

۳-٦ توابع کتابخانه math

همان طوری که می‌دانید، یک کلاس می‌تواند دارای توابع عضوی باشد که سرویس‌های کلاس را انجام می‌دهند. برای مثال، در فصل‌های ۳ الی ۵، توابع عضو نسخه‌های مختلفی از یک شی **Gradebook** را

برای نمایش پیغام خوش‌آمدگویی، تنظیم نام دوره، دستیابی به مجموعه نمرات و محاسبه میانگین این نمرات فراخوانی کرده‌اید.

گاهی اوقات توابع اعضای یک کلاس نیستند. چنین توابعی، توابع سراسری نامیده می‌شوند. نمونه های اولیه تابع در توابع سراسری نیز مانند توابع عضو یک کلاس، در فایلهای سرآیند قرار دارند، از اینروست که توابع سراسری می‌توانند در هر برنامه‌ای که فایل سرآیند مربوطه را ضمیمه می‌کند مورد استفاده مجدد قرار گیرند. برای مثال، به خاطر دارید که از تابع **pow** فایل سرآیند <**cmath**> برای به توان رساندن یک عدد در برنامه شکل ۶-۵ استفاده کردیم. به معرفی توابع مختلف از فایل سرآیند <**cmath**> می‌پردازیم تا مفهوم توابع سراسری را که به یک کلاس خاص تعلق ندارند بیان کنیم. در این فصل و فصلهای آتی، از ترکیبی از توابع سراسری (همانند **main**) و کلاسهای دارای توابع عضو برای پیاده‌سازی مثال‌های خود استفاده خواهیم کرد.

فایل سرآیند <**cmath**> کلکسیونی از توابع تدارک دیده است که به برنامه‌نویس امکان می‌دهند تا بسیاری از محاسبات رایج در ریاضیات را انجام دهد. برای مثال، ممکن است برنامه‌نویس علاقمند به محاسبه و نمایش ریشه دوم 900.0 باشد، از اینرو می‌تواند از عبارت زیر استفاده کند

```
sqrt( 900.0 )
```

زمانیکه این عبارت اجرا شود، تابع **sqrt** فراخوانی می‌شود تا ریشه دوم عدد قرار گرفته در درون پرانتزها (900.0) را محاسبه کند. این تابع آرگومانی از نوع **double** دریافت و نتیجه‌ای از نوع **double** برگشت می‌دهد. دقت کنید که قبل از فراخوانی تابع **sqrt** نیازی به ایجاد هیچ شی نیست. عدد 900.0 آرگومان تابع **sqrt** محسوب می‌شود. عبارت فوق مقدار 30.0 را بدست خواهد داد. همچنین توجه نمائید که تمام توابع موجود در فایل سرآیند <**cmath**> از نوع توابع سراسری هستند و از اینرو، برای فراخوانی کافیست نام تابع و بدنبال آن پرانتزهای حاوی آرگومان مورد نیاز تابع قرار داده شود.

آرگومان‌های تابع می‌توانند مقادیر ثابت، متغیر و حتی عبارات بسیار پیچیده باشند. اگر c=13.0، d=3.0 و f=4.0 باشد، پس عبارت

```
cout << sqrt( c + d * f ) << endl;
```

مبادرت به محاسبه ریشه دوم 25.0 = 4.0 + 3.0 + 13.0 خواهد کرد و پاسخ 5.0 باز می‌گرداند. در جدول شکل ۲-۶ تعدادی از توابع کتابخانه **math** آورده شده است. در این جدول متغیرهای **x** و **y** از نوع **double** هستند.

تابع	توضیح	مثال
fabs(x)	قدر مطلق x	fabs(5.1) is 5.1 fabs(0.0) is 0.0 fabs(-8.76) is 8.76
ceil(x)	گرد کردن x به کوچکترین مقدار صحیح، بطوریکه کوچکتر از x نباشد.	ceil(9.2) is 10.0 ceil(-9.8) is -9.0
cos(x)	محاسبه کسینوس x (x بر حسب رادیان)	cos(0.0) is 1.0
exp(x)	محاسبه e بتوان x	exp(1.0) is 2.71828 exp(2.0) is 7.38906
floor(x)	گرد کردن x به بزرگترین عدد صحیح، بطوریکه بزرگتر از x نباشد.	floor(9.2) is 9.0 floor(-9.8) is -10.0
log(x)	لگاریتم طبیعی x (برپایه e)	log(2.718282) is 1.0 log(7.389056) is 2.0
max(x, y)	مقدار بزرگ x و y	max(2.3, 12.7) is 12.7 max(-2.3, -12.7) is -2.3
min(x, y)	مقدار کوچک x و y	min(2.3, 12.7) is 2.3 min(-2.3, -12.7) is -12.7
pow(x, y)	محاسبه x بتوان y	pow(2.0, 7.0) is 128.0 pow(9.0, .5) is 3.0
round(x)	گرد کردن x به نزدیکترین مقدار صحیح	round(9.75) is 10 round(9.25) is 9
sin(x)	محاسبه سینوس x (x بر حسب رادیان)	sin(0.0) is 0.0
sqrt(x)	محاسبه ریشه دوم x	sqrt(900.0) is 30.0 sqrt(9.0) is 3.0
tan(x)	محاسبه تانژانت x (x برحسب رادیان)	tan(0.0) is 0.0

شکل ۲-۶ |توابع کتابخانه math.

۶-۴ تعریف تابع با پارامترهای مضاعف

در فصل‌های ۳ الی ۵ کلاس‌هایی ارائه شده که حاوی توابع ساده‌ای بودند که حداکثر یک پارامتر داشتند. توابع برای انجام کارهای خود غالبا نیازمند بیش از یک داده هستند. در این بخش به بررسی توابع با پارامترهای مضاعف می‌پردازیم.

برنامه موجود در شکل‌های ۳-۶ الی ۵-۶ کلاس **GradeBook** را با اضافه کردن یک تابع که توسط کاربر تعریف شده، بنام تابع **maximum** اصلاح می‌کند. این تابع، بزرگترین مقدار از میان سه مقدار int را تعیین کرده و برگشت می‌دهد. زمانیکه که اجرای برنامه آغاز می گردد، تابع **main** (خطوط 14-5 در شکل ۵-۶) یک شی از کلاس **GradeBook** را ایجاد می‌کند (خط 8) و تابع عضو **inputGrade** شی را برای گرفتن سه نمره صحیح از کاربر فراخوانی می‌کند (خط 11). در فایل پیاده‌سازی کلاس **GradeBook** (شکل ۴-۶)، خطوط 55-54 تابع عضو **inputGrades** به کاربر اعلان می‌کنند تا سه مقدار **integer** وارد کند. خط 58 تابع عضو **maximum** (در خطوط 75-62) را فراخوانی می‌کند. تابع **maximum** بزرگترین مقدار را مشخص می سازد، سپس فرمان **return** (خط 74) این مقدار را به مکانی که تابع **inputGrades** از آن مکان تابع **maximum** را فراخوانی کرده است، برگشت می‌دهد. سپس تابع عضو **inputGrades** مقدار برگشتی **maximum** را در عضو داده **maximumGrade** ذخیره می‌کند. سپس این مقدار با فراخوانی تابع **displayGradeReport** (خط 12 در شکل ۵-۶) به خروجی ارسال می شود. [نکته: این تابع را **displayGradeReport** نام داده‌ایم، چراکه نسخه های بعدی کلاس **GradeBook** از این تابع برای نمایش یک گزارش کامل از نمرات، شامل نمرات حداکثر و حداقل، استفاده خواهند کرد.] در فصل هفتم، **GradeBook** را به نحوی توسعه می‌دهیم که تعداد دلخواهی از نمرات را پردازش کند.

```
1  // Fig. 6.3: GradeBook.h
2  // Definition of class GradeBook that finds the maximum of three grades.
3  // Member functions are defined in GradeBook.cpp
4  #include <string> // program uses C++ standard string class
5  using std::string;
6
7  // GradeBook class definition
8  class GradeBook
9  {
10 public:
11    GradeBook( string ); // constructor initializes course name
12    void setCourseName( string ); // function to set the course name
13    string getCourseName(); // function to retrieve the course name
14    void displayMessage(); // display a welcome message
15    void inputGrades(); // input three grades from user
16    void displayGradeReport(); // display a report based on the grades
17    int maximum( int, int, int ); // determine max of 3 values
18 private:
19    string courseName; // course name for this GradeBook
20    int maximumGrade; // maximum of three grades
21 }; // end class GradeBook
```

شکل ۳-٦ |فایل سرآیند GradeBook.

```
1   // Fig. 6.4: GradeBook.cpp
2   // Member-function definitions for class GradeBook that
3   // determines the maximum of three grades.
```

```
4   #include <iostream>
5   using std::cout;
6   using std::cin;
7   using std::endl;
8
9   #include "GradeBook.h" // include definition of class GradeBook
10
11  // constructor initializes courseName with string supplied as argument;
12  // initializes maximumGrade to 0
13  GradeBook::GradeBook( string name )
14  {
15     setCourseName( name ); // validate and store courseName
16     maximumGrade = 0; // this value will be replaced by the maximum grade
17  } // end GradeBook constructor
18
19  // function to set the course name; limits name to 25 or fewer characters
20  void GradeBook::setCourseName( string name )
21  {
22     if ( name.length() <= 25 ) // if name has 25 or fewer characters
23        courseName = name; // store the course name in the object
24     else // if name is longer than 25 characters
25     { // set courseName to first 25 characters of parameter name
26        courseName = name.substr( 0, 25 ); // select first 25 characters
27        cout << "Name \"" << name << "\" exceeds maximum length (25).\n"
28           << "Limiting courseName to first 25 characters.\n" << endl;
29     } // end if...else
30  } // end function setCourseName
31
32  // function to retrieve the course name
33  string GradeBook::getCourseName()
34  {
35     return courseName;
36  } // end function getCourseName
37
38  // display a welcome message to the GradeBook user
39  void GradeBook::displayMessage()
40  {
41     // this statement calls getCourseName to get the
42     // name of the course this GradeBook represents
43    cout << "Welcome to the grade book for\n" << getCourseName() << "!\n"
44        << endl;
45  } // end function displayMessage
46
47  // input three grades from user; determine maximum
48  void GradeBook::inputGrades()
49  {
50     int grade1; // first grade entered by user
51     int grade2; // second grade entered by user
52     int grade3; // third grade entered by user
53
54     cout << "Enter three integer grades: ";
55     cin >> grade1 >> grade2 >> grade3;
56
57     // store maximum in member studentMaximum
58     maximumGrade = maximum( grade1, grade2, grade3 );
59  } // end function inputGrades
60
61  // returns the maximum of its three integer parameters
62  int GradeBook::maximum( int x, int y, int z )
63  {
64     int maximumValue = x; // assume x is the largest to start
65
66     // determine whether y is greater than maximumValue
67     if ( y > maximumValue )
68        maximumValue = y; // make y the new maximumValue
69
70     // determine whether z is greater than maximumValue
71     if ( z > maximumValue )
72        maximumValue = z; // make z the new maximumValue
73
74     return maximumValue;
75  } // end function maximum
76
77  // display a report based on the grades entered by user
78  void GradeBook::displayGradeReport()
79  {
80     // output maximum of grades entered
81     cout << "Maximum of grades entered: " << maximumGrade << endl;
82    } // end function displayGradeReport
```

شکل ٤-٦ |کلاس GradeBook تعریف کننده تابع maximum.

```
1  // Fig. 6.5: fig06_05.cpp
2  // Create GradeBook object, input grades and display grade report.
3  #include "GradeBook.h" // include definition of class GradeBook
4
5  int main()
6  {
7     // create GradeBook object
8     GradeBook myGradeBook( "CS101 C++ Programming" );
9
10    myGradeBook.displayMessage(); // display welcome message
11    myGradeBook.inputGrades(); // read grades from user
12    myGradeBook.displayGradeReport(); // display report based on grades
13    return 0; // indicate successful termination
14 } // end main
```

```
Welcome to the grade book for
CS101 C++ Programming!


Welcome to the grade book for
CS101 C++ Programming!

Welcome to the grade book for
CS101 C++ Programming!

Enter three integer grades: 67 75 86
Maximum of grades entered: 86
```

شکل ٥-٦ |عملکرد تابع maximum.

مهندسی نرم‌افزار

کاماهایی که در خط 58از شکل ۴-۶ برای جدا کردن آرگومان‌های تابع ***maximum*** *مورد استفاده قرار گرفته‌اند با عملگرهای کاما که در بخش۳-۵ توضیح داده شده‌اند یکسان نیستند. عملگر کاما تضمین می‌کند که عملوندهای آن از چپ به راست ارزیابی می‌شوند. اما ترتیب ارزیابی آرگومان‌های یک تابع، از سوی* ***C++*** *مشخص نمی‌شود. از اینرو، کامپایلرهای مختلف می‌توانند آرگومان‌های تابع را به ترتیب‌های متفاوت ارزیابی کنند.*

قابلیت حمل

گاهی اوقات زمانی که آرگومان‌های یک تابع عبارات بیشتری را شامل می‌شوند، همانند توابعی که توابع دیگری را فراخوانی می‌کنند، ترتیب ارزیابی آرگومان‌ها از سوی کامپایلر می‌تواند مقادیر یک یا چند آرگومان را تحت تاثیر قرار دهد. اگر ترتیب ارزیابی در میان کامپایلرها متفاوت باشد، مقادیر آرگومان ارسال شده به تابع می‌تواند تغییر پیدا کند، که این حالت خطاهای منطقی بوجود می‌آورد.

اجتناب از خطا

اگر در مورد ترتیب ارزیابی آرگومان‌های یک تابع و ترتیب ارزیابی مقادیر ارسالی به تابع شک دارید، آرگومان‌ها را قبل از فراخوانی تابع درعبارات تخصیصی مجزا مورد ارزیابی قرار داده، و نتیجه هر عبارت را به یک متغیر محلی تخصیص دهید، سپس این متغیرها را به عنوان آرگومان به تابع ارسال کنید.

نمونه اولیه تابع عضو **maximum** (شکل ۳-۶، خط 17) تعیین می‌کند که تابع یک مقدار صحیح برگشت می‌دهد، نام تابع **maximum** است و تابع برای انجام کار خود به سه پارامتر صحیح نیاز دارد. سرآیند تابع

maximum (شکل ۴-۶، خط 62) با نمونه اولیه تابع مطابقت دارد و نشان می‌دهد که پارامترها **x**، **y** و **z** نام دارند. زمانیکه **maximum** فراخوانی شود (شکل ۴-۶، خط 58)، پارامتر **x** با مقدار آرگومان **grade1**، پارامتر **y** با مقدار آرگومان **grade2** و پارامتر **z** با مقدار آرگومان **garde3** مقداردهی اولیه می‌شوند. فراخوانی تابع بر اساس تعریف تابع باید برای هر پارامتر یک آرگومان داشته باشد.

دقت کنید که هم در نمونه اولیه تابع و هم در سرآیند تابع چندین پارامتر به صورت یک لیست که توسط کاما از هم جدا شده‌اند، مشخص شده است. کامپایلر برای بررسی این مطلب که آیا فراخوانی‌های تابع **maximum** تعداد و نوع درستی از آرگومان‌ها را دارد و نوع آرگومان‌ها به ترتیب صحیحی در کنار هم قرار گرفته‌اند به نمونه اولیه تابع مراجعه می‌کند. علاوه بر این، کامپایلر نمونه اولیه را برای اطمینان از درستی مقدار برگشتی تابع به عبارتی که تابع را فراخوانی کرده است مورد استفاده قرار می‌دهد (برای مثال فراخوانی تابعی که **void** را برگشت می‌دهد، نمی‌تواند در سمت راست یک دستور تخصیص بکار گرفته شود). هر آرگومان باید با نوع پارامتر متناظرش سازگار باشد. برای مثال، یک پارامتر از نوع **double** می‌تواند مقادیری از قبیل 7.37، 22 یا 0.03456- را دریافت کند، اما قادر به دریافت رشته‌ای همانند "hello" نیست. اگر آرگومان‌های ارسالی به یک تابع با نوع‌های تعیین شده در نمونه اولیه تابع مطابقت نداشته باشند، کامپایلر آرگومان‌ها را به نوع متناظر تبدیل می‌کند. در بخش ۵-۶ این تبدیل توضیح داده شده است.

خطای برنامه‌نویسی

*اعلان پارامترهای متد از یک نوع بصورت **double x، y** به جای **double x، double y** خطای نحوی است. چرا که برای هر پارامتر در لیست پارامتری باید یک نوع صریح تعریف شود.*

خطای برنامه‌نویسی

اگر نمونه اولیه تابع، سرآیند تابع و فراخوانی‌های تابع همگی از نظر تعداد، نوع و ترتیب آرگومان‌ها و پارامترها، و از نظر نوع برگشتی مطابقت نداشته باشند، خطای کامپایل رخ خواهد داد.

مهندسی نرم‌افزار

تابعی که دارای تعداد زیادی پارامتر است احتمالا وظایف زیادی دارد. تقسیم تابع به توابع کوچکتر که هر یک وظیفه خاصی را انجام می‌دهند، می‌تواند سرآیند تابع را به یک خط محدود کند.

برای تعیین مقدار ماکزیمم (خطوط 75-62 از شکل ۴-۶)، با این فرض کار را آغاز می‌شود که پارامتر **x** حاوی بزرگترین مقدار است، از اینرو خط 64 در تابع **maximum** متغیر محلی **maximumValue** را اعلان کرده و آن را با مقدار پارامتر **x** مقداردهی اولیه می‌کند. البته این امکان وجود دارد که پارامتر **y** یا **z** حاوی بزرگترین مقدار باشند، بنابر این باید هر یک از این مقادیر را با **maximumValue** مقایسه کنیم. عبارت **if** در خطوط 68-67 تعیین می‌کند که آیا **y** بزرگتر از **maximumValue** است یا خیر، اگر چنین

باشد، **y** را به **maximumValue** تخصیص می‌دهد. عبارت **if** در خطوط 72-71 تعیین می‌کند که آیا **z** بزرگتر از **maximumValue** است یا خیر، اگر چنین باشد، **z** را به **maximumValue** تخصیص می‌دهد. در این مرحله بزرگترین مقدار در **maximumValue** قرار دارد، بنابر این خط 74 این مقدار را به فراخوان در خط 58 برگشت می‌دهد. زمانیکه کنترل برنامه به نقطه‌ای از برنامه که **maximum** فراخوانی شده باز می‌گردد، پارامترهای **maximum** یعنی **x**، **y**،**z** دیگر برای برنامه دسترس پذیر نیستند. در بخش بعد به بررسی این مسئله خواهیم پرداخت.

سه روش برای بازگرداندن کنترل به نقطه‌ای که تابع فراخوانی شده است وجود دارد. اگر تابع نتیجه‌ای برگشت ندهد (نوع برگشتی تابع **void** باشد)، کنترل زمانیکه برنامه به انتهای تابع (براکت سمت راست) یا عبارت **return;** برسد، برگشت داده خواهد شد. اگر تابع مقداری برگشت دهد، عبارت

```
return عبارت;
```

مقدار *عبارت* را به فراخوان برگشت می‌دهد. هنگامی که عبارت **return** اجرا می‌شود، بلافاصله کنترل به نقطه‌ای که تابع از آن مکان فعال شده، برگشت داده می‌شود.

٦-٥ نمونه اولیه تابع و تبدیل آرگومان

نمونه اولیه یک تابع (که اعلان یک تابع هم نامیده می‌شود) به کامپایلر نام تابع، نوع داده برگشتی تابع، تعداد پارامترهایی که تابع انتظار دریافت آنها را دارد، نوع و ترتیب این پارامترها را اعلان می‌کند.

مهندسی نرم‌افزار

*نمونه‌های اولیه تابع در **C++** الزامی هستند. از دستور دهنده‌های پیش پردازنده **#include** برای دستیابی به نمونه‌های اولیه تابع موجود در فایل‌های سرآیند در کتابخانه‌های مناسب (همانند، نمونه اولیه برای تابع ریاضی **sqrt** در فایل سرآیند <**cmath**>) برای توابع کتابخانه استاندارد **C++** استفاده کنید. همچنین از **#include** برای دستیابی به فایلهای سرآیند حاوی نمونه‌های اولیه که توسط خودتان یا اعضای گروه نوشته شده، استفاده کنید.*

خطای برنامه‌نویسی

اگر تابعی قبل از آنکه فراخوانی شود تعریف شده باشد، پس تعریف تابع به عنوان نمونه اولیه تابع عمل خواهد کرد، از اینرو نیازی به یک نمونه اولیه مجزا نیست. اگر یک تابع قبل از آنکه تعریف شود فراخوانی گردد و دارای یک نمونه اولیه تابع نباشد، خطای کامپایل رخ خواهد داد.

مهندسی نرم‌افزار

همیشه نمونه‌های اولیه تابع را تدارک ببینید، حتی اگر توابع پیش از آنکه مورد استفاده قرار گیرند تعریف شده باشند (در حالتی که سرآیند تابع به عنوان نمونه اولیه تابع هم عمل کند) تدارک دین نمونه های اولیه از گره خوردن کد به تعریف ترتیب توابع جلوگیری می‌کند.

امضاء تابع

قسمتی از نمونه اولیه یک تابع که شامل نام تابع و نوع آرگومان‌های آن است بعنوان "امضای تابع" یا "امضا" شناخته می شود. امضای تابع نوع برگشتی تابع را تعیین نمی‌کند. توابع موجود در یک قلمرو باید دارای امضاهای منحصر به فرد باشند. قلمرو یک تابع منطقه‌ای از برنامه است که تابع در آن شناخته شده و در دسترس می‌باشد. در بخش ۱۰-۶ به بررسی دقیق‌تر قلمرو پرداخته‌ایم.

خطای برنامه‌نویسی

اگر دو تابع در یک قلمرو دارای امضاهای یکسان باشند اما نوع برگشتی آنها متفاوت باشد، با خطای کامپایل مواجه خواهید شد.

در شکل ۳-۶ اگر نمونه اولیه تابع در خط 17 به اینصورت نوشته شده بود

```
void maximum( int, int, int );
```

کامپایلر خطا گزارش می‌کرد، چرا که نوع برگشتی **void** در نمونه اولیه تابع با نوع برگشتی **int** در سرآیند تابع متفاوت است. به همین ترتیب، چنین نمونه اولیه سبب خواهد شد، عبارت

```
cout << maximum( 6, 9, 0 );
```

خطای کامپایل تولید کند، چرا که عبارت فوق وابسته به **maximum** برای برگشت دادن مقدار برای نمایش است.

الزام یا تبدیل آرگومان

یکی از ویژگی‌های مهم در نمونه اولیه توابع، *تبدیل آرگومان* است (مجبور کردن آرگومان‌ها برای بدست آوردن نوع داده مقتضی مشخص شده در اعلان پارامترها). برای مثال، برنامه‌ای می‌تواند یک تابع با یک آرگومان از نوع صحیح را فراخوانی کند، حتی اگر نمونه اولیه تابع آرگومان را از نوع **double** مشخص کرده باشد. در اینحالت هم برنامه بدرستی کار خواهد کرد.

قوانین ترقی آرگومان

گاهی اوقات، مقادیر آرگومان که دقیقا با نوع‌های پارامتر در نمونه اولیه تابع مطابقت ندارند، می‌توانند قبل از آنکه تابع فراخوانی شود توسط کامپایلر به نوع مناسب تبدیل گردند. این تبدیل‌ها تحت قوانینی بنام قوانین ترقی آرگومان در C++، رخ می‌دهند. قوانین ترقی تعیین می‌کنند که چگونه می‌توان بدون از دست دادن داده‌ها به تبدیل نوع اقدام کرد. یک **int** بدون تغییر یافتن مقدار خود، می‌تواند به یک نوع **double** تبدیل گردد. ولی در تبدیل یک **double** به یک **int**، قسمت اعشاری مقدار **double** از دست خواهد رفت. به خاطر دارید که متغیرهای **double** می‌توانند اعدادی با ارقام بسیار بیشتر از اعداد **int** در خود ذخیره کنند، از اینرو امکان از رفتن داده وجود دارد. همچنین امکان دارد که مقادیر در ضمن تبدیل

نوع‌های عددی بزرگتر به نوع‌های عددی کوچکتر (همانند **long** به **short**)، علامت‌دار به بدون علامت یا بدون علامت به علامت‌دار تغییر یابند.

قوانین ترقی بر روی عباراتی که حاوی مقادیری از یک نوع داده یا چندین نوع داده باشند، اعمال می‌شود، چنین عباراتی به عنوان عبارات ترکیبی شناخته می‌شوند. نوع هر مقدار موجود در یک عبارت نوع ترکیبی به "بالاترین" نوع موجود در عبارت ارتقا می‌یابد (در واقع یک نسخه موقت از هر مقدار ایجاد شده و برای عبارت بکار گرفته می‌شود، مقادیر اصلی بدون تغییر باقی می‌مانند). همچنین ترقی زمانی رخ می‌دهد که نوع آرگومان یک تابع با نوع پارامتر تعیین شده در تعریف تابع یا نمونه اولیه تابع مطابقت نداشته باشد. در جدول شکل ۶-۶ نوع‌های داده بنیادین به ترتیب از "بالاترین نوع" به "پایین‌ترین نوع" لیست شده‌اند.

تبدیل مقادیر به نوع های بنیادین پایین‌تر می‌تواند منجر به تولید مقادیر اشتباه گردد. از اینرو، یک مقدار فقط در صورتی می‌تواند به یک نوع بنیادین پایین‌تر تبدیل شود که به فرم صریح به یک متغیر از نوع پایین‌تر تخصیص داده شود (تعدادی کامپایلرها در اینحالت هشدار صادر می‌کنند) یا از عملگر تبدیل **cast** به این منظور استفاده شود (به بخش ۹-۴ مراجعه کنید). نحوه تبدیل مقادیر آرگومان تابع به نوع های پارامتر در نمونه اولیه یک تابع دقیقا همانند تخصیص دادن مقادیر به متغیرهایی از آن نوع است. اگر تابع **square** که از یک پارامتر عددی از نوع صحیح استفاده می‌کند با آرگومان اعشاری فراخوانی شود، آرگومان به **int** تبدیل می‌شود(نوع پایین تر) و احتمالا **square** مقدار نادرستی برگشت خواهد داد. برای مثال، **square(4.5)** مقدار 16 را برگشت می‌دهد و نه مقدار20.25.

نوع داده
long double
double
float
unsigned long int
long int
unsigned int
int
unsigned short int
short int

```
unsigned char
```

```
char
```

```
bool
```

شکل ٦-٦ | سلسله مراتب ترقی نوع‌های بنیادین.

خطای برنامه‌نویسی

به هنگام تبدیل از نوع داده بالاتر در سلسله مراتب ترقی به یک نوع پایین‌تر، یا مابین نوع‌های علامت‌دار و بدون علامت، امکان از دست رفتن داده وجود دارد.

خطای برنامه‌نویسی

اگر آرگومان‌های موجود در فراخوانی یک تابع با تعداد و نوع پارامترهای اعلان شده در نمونه اولیه تابع متناظر مطابقت نداشته باشند، خطای کامپایل رخ خواهد داد. همچنین اگر تعداد آرگومانهای در فراخوانی‌ها مطابقت داشته باشند، اما نتوان آرگومان‌ها را بصورت ضمنی به نوع‌های مورد نظر تبدیل کرد، با خطا مواجه خواهید شد.

٦-٦ فایل‌های سرآیند کتابخانه استاندارد C++

کتابخانه استاندارد C++ به قسمت‌های بسیاری تقسیم شده، که هر قسمت دارای فایل سرآیند متعلق به خود است. فایل‌های سرآیند حاوی نمونه‌های اولیه تابع برای توابع مرتبطی است که هر بخش از کتابخانه را تشکیل می‌دهند. همچنین فایل‌های سرآیند حاوی تعاریفی از انواع کلاس و توابع به همراه ثابت‌های مورد نیاز این توابع می‌باشند. یک فایل سرآیند کامپایلر را در مورد نحوه برقراری ارتباط با کامپونت‌های کتابخانه و نوشته شده توسط کاربر هدایت می‌کند.

در جدول شکل ٦-٧ تعدادی از فایل‌های سرآیند رایج از کتابخانه استاندارد C++ لیست شده است که در مورد اکثر آنها در این کتاب صحبت خواهد شد. اسامی فایل‌های سرآیند که با **.h** پایان می‌یابند فایل‌های سرآیند از نوع سبک قدیمی هستند که توسط فایل‌های سرآیند کتابخانه استاندارد C++ جایگزین گشته‌اند. در این کتاب فقط از نسخه‌های کتابخانه استاندارد C++ از هر فایل سرآیند استفاده کرده‌ایم تا مطمئن شویم که مثال‌های مطرح شده بر روی اکثر کامپایلرهای استاندارد C++ عمل خواهند کرد.

فایل‌های سرآیند کتابخانه استاندارد C++	توضیحات
<iostream>	حاوی نمونه‌های اولیه تابع برای توابع ورودی و خروجی استاندارد C++ است که در فصل دوم معرفی شده است، و در فصل پانزدهم جزئیات بیشتری از آن بررسی خواهد شد. این فایل سرآیند جایگزین فایل سرآیند <iostream.h> شده است.
<iomanip>	حاوی نمونه‌های اولیه تابع برای دستکاری کننده‌های جریان است که جریان‌های داده را

	قالب‌بندی می‌کنند. از این فایل سرآیند ابتدا در بخش ۹-۴ استفاده شده است و در فصل پانزدهم جزئیات بیشتری از آن بررسی خواهد شد. این فایل سرآیند جایگزین فایل سرآیند <**iomanip.h**> شده است
<cmath>	حاوی نمونه‌های اولیه تابع برای توابع کتابخانه‌ای ریاضی است. این فایل سرآیند جایگزین فایل سرآیند<**math.h**> شده است.
<cstdlib>	حاوی نمونه‌های اولیه تابع به منظور تبدیل اعداد به متن، متن به عدد، تخصیص حافظه، اعداد تصادفی و برخی از توابع یوتیلیتی یا کمکی دیگر است. قسمت‌های از این فایل سرآیند در بخش ۷-۶، فصل یازدهم، فصل شانزدهم، فصل نوزدهم مورد بررسی قرار گرفته‌اند. این فایل سرآیند جایگزین فایل سرآیند<**stdlib.h**> شده است.
<ctime>	حاوی نمونه‌های اولیه و نوع‌های تابع برای مدیریت زمان و تاریخ است. این فایل سرآیند جایگزین فایل سرآیند<time.h> شده است. این فایل سرآیند در بخش ۷-۶ بکار رفته است.
<vector>,<list>, <deque>,<queue>, <stack>,<map>, <set>,<bitset>	این فایل‌های سرآیند حاوی کلاس‌هایی هستند که حامل‌های کتابخانه استاندارد C++ را پیاده‌سازی می‌کنند. حامل‌ها مبادرت به ذخیره‌سازی داده‌ها در طی اجرای یک برنامه می‌کنند. سرآیند <**vector**> ابتدا در فصل هفتم معرفی خواهد شد.
<cctype>	حاوی نمونه‌های اولیه تابع برای توابعی است که کاراکترها را برای خصوصیت‌های خاصی تست می‌کنند (همانند اینکه آیا کاراکتر یک رقم است یا یک نقطه‌گذاری)، و نمونه‌های اولیه تابع برای توابعی است که می‌توانند در تبدیل حروف‌کوچک به حروف‌بزرگ و بالعکس بکار گرفته شوند. این فایل سرآیند جایگزین فایل سرآیند <**ctype.h**> شده است. این مباحث در فصل هشتم مطرح شده‌اند.
<cstring>	حاوی نمونه‌های اولیه تابع برای توابع پردازش رشته به سبک C است و این فایل سرآیند جایگزین فایل سرآیند <**string.h**> شده است. در فصل یازدهم از این سرآیند استفاده شده.
<typeinfo>	حاوی کلاس‌هایی برای شناسایی نوع در زمان اجرا (تعیین نوع داده در زمان اجرا) است. این فایل سرآیند در بخش ۸-۱۳ بکار گرفته شده است.
<exception>, <stdexcept>	این فایل‌های سرآیند حاوی کلاس‌هایی هستند که در مدیریت استثناء بکار می‌روند.
<memory>	حاوی کلاس‌ها و توابعی است که از طرف کتابخانه استاندارد C++ به منظور تخصیص حافظه برای حامل‌های کتابخانه استاندارد C++ بکار گرفته می‌شوند. از این سرآیند در فصل شانزدهم استفاده شده است.
<fstream>	حاوی نمونه‌های اولیه تابع برای توابعی است که عملیلت ورودی از فایل‌های موجود بر روی دیسک و خروجی به فایل‌های موجود بر روی دیسک (در فصل هفدهم بکار گرفته شده‌اند) را انجام می‌دهند. این فایل سرآیند جایگزین فایل سرآیند<**fstream.h**> شده است.
<string>	حاوی تعریف کلاس **string** از کتابخانه استاندارد C++ است(در فصل هیجدهم بکار گرفته شده است).
<sstream>	حاوی نمونه‌های اولیه تابع برای توابعی است که عملیات ورودی از رشته‌های موجود در

	حافظه را پیاده‌سازی می‌کنند (در فصل هیجدهم بکار گرفته شده است).
<functional>	حاوی کلاس‌ها و توابعی است که از طرف الگوریتم‌های کتابخانه استاندارد C++ بکار برده می‌شوند.
<iterator>	حاوی کلاس‌هایی برای دسترسی به داده‌ها در حامل‌های کتابخانه استاندارد C++ است.
<algorithm>	حاوی توابعی برای مدیریت داده‌ها در حامل‌های کتابخانه استاندارد C++ است.
<cassert>	حاوی ماکروهایی برای تشخیص خطاها در برنامه است. این فایل سرآیند جایگزین فایل سرآیند <**assert.h**> در C++ قبل از استاندارد شده است.
<cfloat>	حاوی محدودیت سایز اعشاری سیستم است. این فایل سرآیند جایگزین سرآیند<**float.h**> شده است.
<climits>	حاوی محدودیت سایز اعداد صحیح سیستم است. این فایل سرآیند جایگزین فایل سرآیند <limits.h> شده است.
<cstdio>	حائی نمونه‌های اولیه تابع برای توابع کتابخانه ورودی/خروجی استاندارد سبک C و اطلاعات بکار رفته توسط آنها است. این فایل سرآیند جایگزین فایل سرآیند <**stdio.h**> شده است.
<locate>	حاوی کلاس‌ها و توابعی است که معمولا از طرف پردازش جریان برای پردازش داده‌ها به شکل طبیعی برای زبان‌های مختلف (همانند، فرمت‌های پولی، مرتب‌سازی رشته‌ها، عرضه کاراکترها و ...) بکار گرفته می‌شود.
<limits>	حاوی کلاس‌هایی برای تعریف محدودیت‌های نوع داده بر روی پلاتفرم هر کامپیوتری است.
<utility>	حاوی کلاس‌ها و توابعی است که توسط بسیاری از فایل‌های سرآیند کتابخانه استاندارد C++ بکار گرفته می‌شوند.

شکل ۷-۶ | فایل‌های سرآیند کتابخانه استاندارد C++.

۶-۷ مبحث آموزشی: تولید اعداد تصادفی

در این بخش به بحث برنامه‌نویسی برنامه‌های بازی و شبیه سازی می‌پردازیم. در این بخش و بخش بعدی، با استفاده از ساختارهای کنترلی که قبلاً با آنها آشنا شده‌اید یک برنامه بازی ایجاد خواهیم کرد که حاوی توابع متعددی است. این بازی در ارتباط با شانس است. عنصر شانس می‌تواند در برنامه‌های کامپیوتری از طریق تابع کتابخانه استاندارد ***rand*** ایجاد شود.

به عبارت زیر توجه نمائید:

```
i = rand();
```

تابع **rand** یک مقدار صحیح بدون علامت مابین صفر و ثابت **RAND_MAX** ایجاد می‌کند. یک ثابت نمادین ایجاد شده در فایل سرآیند <**cstdlib**>. بایستی مقدار **RAND_MAX** حداقل 32767 باشد، حداکثر مقدار مثبت برای یک عدد صحیح دو بایتی (16 بیت). در GNU C++، مقدار **RAND_MAX**

برابر با 214748647 و در ویژوال استودیو این مقدار برابر 32767 است. اگر **rand** مقادیری بصورت تصادفی ایجاد کند، هر مقدار در این محدودهٔ در هر بار فراخوانی تابع **rand** دارای *شانس* (احتمال) برابر خواهد بود.

گاهاً ایجاد اعداد تصادفی در یک برنامه ضرورت پیدا می‌کند. با این وجود، محدودهٔ مقادیر تولید شده توسط **rand** غالباً متفاوت از مقدار مورد نیاز در یک برنامه هستند. برای مثال، در برنامه‌ای که پرتاب را شبیه‌سازی می‌کند، فقط نیاز به مقدار 0 برای نشان دادن "رو" و 1 برای "پشت" سکه نیاز دارد، یا برنامه‌ای که پرتاب یک طاس شش وجهی را شبیه‌سازی می‌کند، نیاز به مقادیر تصادفی از 1 تا 6 دارد. به همین ترتیب، برنامه‌ای که حرکت یک سفینه فضایی را تداعی می‌کند و نیاز به حرکت در چهار جهت را دارد، مستلزم بدست آوردن عدد تصادفی از 1 تا 4 است.

پرتاب طاس شش وجهی

برای توصیف **rand**، اجازه دهید برنامه‌ای ایجاد کنیم (شکل ۸-۶) که 20 پرتاب یک تاس شش وجهی و چاپ مقدار هر پرتاب را شبیه‌سازی نماید. نمونه اولیه تابع **rand** در سرآیند <**cstdlib**> قرار دارد. برای تولید اعداد صحیح در بازه 0 تا 5، از عملگر باقیمانده (**%**) به همراه **rand** استفاده می‌کنیم:

```
rand() % 6
```

این عمل بعنوان تغییر مقیاس شناخته می‌شود. عدد 6 فاکتور تغییر مقیاس نامیده می‌شود. سپس بازه اعداد تولیدی را با افزودن عدد 1 به نتیجه قبلی، جابجا یا شیفت می‌دهیم. برنامه شکل ۸-۶ نشان می‌دهد که نتایج در بازه 1 تا 6 قرار دارند.

```
1  // Fig. 6.8: fig06_08.cpp
2  // Shifted and scaled random integers.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <iomanip>
8  using std::setw;
9
10 #include <cstdlib> // contains function prototype for rand
11 using std::rand;
12
13 int main()
14 {
15    // loop 20 times
16    for ( int counter = 1; counter <= 20; counter++ )
17    {
18       // pick random number from 1 to 6 and output it
19       cout << setw( 10 ) << ( 1 + rand() % 6 );
20
21       // if counter is divisible by 5, start a new line of output
22       if ( counter % 5 == 0 )
23          cout << endl;
24    } // end for
25
26    return 0; // indicates successful termination
27 } // end main
```

```
 6          6          5          5          6
 5          1          1          5          3
 6          6          2          4          2
 6          2          3          4          1
```

شکل ۸-٦ | اعداد تصادفی در محدوده 1-6.

شش میلیون بار پرتاب یک طاس شش وجهی

برای اینکه نشان دهیم اعداد تولید شده توسط تابع **rand** تقریبا با احتمال برابر رخ می‌دهند، برنامه شکل ۹-۶ شش میلیون پرتاب یک تاس را شبیه‌سازی می‌کند. هر عدد صحیح در بازه 1 تا 6 باید تقریبا یک میلیون بار ظاهر گردد. این حالت در پنجره خروجی شکل ۹-۶ آورده داده شده است.

```
1  // Fig. 6.9: fig06_09.cpp
2  // Roll a six-sided die 6,000,000 times.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <iomanip>
8  using std::setw;
9
10 #include <cstdlib> // contains function prototype for rand
11 using std::rand;
12
13 int main()
14 {
15    int frequency1 = 0; // count of 1s rolled
16    int frequency2 = 0; // count of 2s rolled
17    int frequency3 = 0; // count of 3s rolled
18    int frequency4 = 0; // count of 4s rolled
19    int frequency5 = 0; // count of 5s rolled
20    int frequency6 = 0; // count of 6s rolled
21
22    int face; // stores most recently rolled value
23
24    // summarize results of 6,000,000 rolls of a die
25    for ( int roll = 1; roll <= 6000000; roll++ )
26    {
27       face = 1 + rand() % 6; // random number from 1 to 6
28
29       // determine roll value 1-6 and increment appropriate counter
30       switch ( face )
31       {
32          case 1:
33             ++frequency1; // increment the 1s counter
34             break;
35          case 2:
36             ++frequency2; // increment the 2s counter
37             break;
38         case 3:
39             ++frequency3; // increment the 3s counter
40             break;
41          case 4:
42             ++frequency4; // increment the 4s counter
43             break;
44          case 5:
45             ++frequency5; // increment the 5s counter
46             break;
47          case 6:
48             ++frequency6; // increment the 6s counter
49             break;
50          default: // invalid value
51             cout << "Program should never get here!";
52       } // end switch
53    } // end for
54
55   cout << "Face" << setw( 13 ) << "Frequency" << endl; // output headers
56    cout << "   1" << setw( 13 ) << frequency1
57       << "\n   2" << setw( 13 ) << frequency2
58       << "\n   3" << setw( 13 ) << frequency3
59       << "\n   4" << setw( 13 ) << frequency4
60       << "\n   5" << setw( 13 ) << frequency5
61       << "\n   6" << setw( 13 ) << frequency6 << endl;
62    return 0; // indicates successful termination
63 } // end main
```

```
Face        Frequency
   1           999702
   2          1000823
   3           999378
   4           998898
   5          1000777
   6          1000422
```

شکل ۹-٦ | پرتاپ 6,000,000 بار طاس شش وجهی.

همانطوری که خروجی برنامه نشان می‌دهد، می‌توانیم پرتاب یک تاس شش وجهی را با تغییر دادن مقیاس و شیفت مقادیر تولید شده از طرف **rand** شبیه‌سازی نمائیم. دقت کنید که برنامه هرگز نباید به حالت **default** (خطوط 51-50) در ساختار **switch** وارد گردد، برای اینکه عبارت کنترلی **switch** یعنی **face** همیشه در بازه 1-6 قرار دارد، با این وجود حالت **default** را بعنوان یک تمرین و عمل خوب در نظر گرفته‌ایم. پس از آنکه با آرایه‌ها در فصل هفتم آشنا گردیدید، با نحوه جایگزین ساختن کل ساختار **switch** بکار رفته در شکل ۹-۶ با یک عبارت آشنا خواهید شد.

تصادفی کردن تولید اعداد تصادفی

با اجرای برنامه شکل ۸-۶ مجددا این مقادیر تولید می‌شوند

```
6       6       5       5       6
5       1       1       5       3
6       6       2       4       2
6       2       3       4       1
```

توجه کنید که برنامه دقیقا همان دنباله از مقادیر نشان داده شده در شکل ۸-۶ را چاپ می‌کند. این مقادیر چگونه می‌توانند تصادفی باشند؟ بطور کلی، این قابلیت تکرار یکی از صفات مهم تابع **rand** است. به هنگام دیباگ یک برنامه شبیه‌سازی، این خصیصه تکرار برای اثبات اینکه اصلاحات صورت گرفته در برنامه به درستی عمل می‌کنند، لازم است.

در واقع تابع **rand** اعداد شبه‌تصادفی تولید می‌کند. فراخوانی مکرر **rand** دنباله‌ای از اعداد تولید می‌کند که به نظر تصادفی می‌رسند. با این وجود، خود این توالی هر دفعه که برنامه اجرا می‌شود تکرار می‌گردد. زمانیکه برنامه کاملا خطایابی شد، می‌توان در هر بار اجرا دنباله متفاوتی از اعداد تصادفی تولید کرد. این فرآیند بعنوان تصادفی‌سازی مطرح است و با استفاده از تابع **srand** کتابخانه استاندارد C++ صورت می‌گردد. تابع **srand** یک آرگومان صحیح بدون علامت دریافت و تابع **rand** را برای تولید دنباله متفاوتی از اعداد تصادفی در هر بار اجرای برنامه *تغذیه* یا ***seed*** می‌نماید.

برنامه شکل ۱۰-۶ عملکرد تابع **srand** را نشان می‌دهد. برنامه از نوع داده **unsigned** استفاده می‌کند، که شکل کوتاه شده **unsigned int** است. یک **int** حداقل در دو بایت از حافظه ذخیره می‌شود (عموما چهار بایت از حافظه در سیستم‌های 32 بیتی) و می‌تواند حاوی مقادیر مثبت و منفی باشد. یک متغیر از نوع **unsigned int** حداقل در دو بایت از حافظه ذخیره می‌شود. یک **unsigned int** دو بایتی می‌تواند فقط دارای مقادیر مثبت در بازه 0-65535 باشد. یک **unsigned int** چهار بایتی می‌تواند فقط دارای مقادیر

مثبت در بازه 0-4294967295 باشد. تابع **srand** یک مقدار **unsigned int** را بعنوان آرگومان دریافت می‌کند. نمونه اولیه تابع برای **srand** در فایل سرآیند <**cstdlib**> قرار دارد.

```
1  // Fig. 6.10: fig06_10.cpp
2  // Randomizing die-rolling program.
3  #include <iostream>
4  using std::cout;
5  using std::cin;
6  using std::endl;
7
8  #include <iomanip>
9  using std::setw;
10
11 #include <cstdlib> // contains prototypes for functions srand and rand
12 using std::rand;
13 using std::srand;
14
15 int main()
16 {
17    unsigned seed; // stores the seed entered by the user
18
19    cout << "Enter seed: ";
20    cin >> seed;
21    srand( seed ); // seed random number generator
22
23    // loop 10 times
24    for ( int counter = 1; counter <= 10; counter++ )
25    {
26       // pick random number from 1 to 6 and output it
27       cout << setw( 10 ) << ( 1 + rand() % 6 );
28
29       // if counter is divisible by 5, start a new line of output
30       if ( counter % 5 == 0 )
31          cout << endl;
32    } // end for
33
34    return 0; // indicates successful termination
35 } // end main
```

```
Enter seed: 67
         6         1         4         6         2
         1         6         1         6         4
```

```
Enter seed: 432
         4         6         3         1         6
         3         1         5         4         2
```

```
Enter seed: 67
         6         1         4         6         2
         1         6         1         6         4
```

شکل ۱۰-٦ | تصادفی کردن پرتاب طاس شش وجهی.

اجازه دهید تا برنامه را چندین بار اجرا کرده و به بررسی نتایج بپردازیم. دقت کنید که برنامه در هر بار اجرا دنباله متفاوتی از اعداد تصادفی را تولید می‌نماید، چرا که کاربر در هر بار یک seed متفاوتی را وارد می‌کند. در خروجی‌های نمونه اول و سوم از seed یکسانی استفاده شده است، از اینرو در هر دو خروجی، دنباله یکسانی از اعداد نمایش داده شده است. برای تصادفی کردن بدون نیاز به وارد کردن یک seed در هر بار، می‌توانیم از عبارتی همانند

```
srand( time( 0 ) );
```

استفاده کنیم. این عبارت سبب می‌شود تا کامپیوتر مبادرت به خواندن ساعت (زمان) خود کرده و مقدار seed را بدست آورد. تابع **time** (با آرگومان **0** که در عبارت فوق نوشته شده است) زمان جاری را به صورت تعداد ثانیه‌های سپری شده از نیمه شب اول ژانویه 1970 به وقت GMT برگشت می‌دهد. این مقدار به یک عدد صحیح بدون علامت تبدیل شده و بعنوان seed در تولید کننده عدد تصادفی بکار گرفته می‌شود. نمونه اولیه تابع برای **time** در <**ctime**> قرار دارد.

خطای برنامه‌نویسی

فراخوانی تابع ***srand*** بیش از یک بار در یک برنامه، مجددا تولید دنباله اعداد شبه‌تصادفی را از ابتدا آغاز می‌کند و می‌تواند تصادفی بودن اعداد تولید شده توسط ***rand*** را تحت تاثیر قرار دهد.

تعمیم ضریب پیمایش و تغییر مکان اعداد تصادفی

قبل از این، در ارتباط با نحوه نوشتن یک دستور واحد برای شبیه سازی پرتاب یک تاس شش وجهی با عبارت زیر توضیحاتی بیان کردیم

```
face = 1 + rand() % 6;
```

که همیشه یک عدد صحیح را به صورت تصادفی به متغیر **face** در محدوده ی $1 \le face \le 6$ تخصیص می‌دهد. توجه کنید که پهنای این محدوده (یعنی تعداد اعداد صحیح متوالی موجود در محدوده) 6 بوده و عدد آغازین در دنباله 1 می‌باشد. با مراجعه به دستور فوق، مشاهده می‌کنید که پهنای محدوده به وسیله عدد بکار رفته در مقیاس **rand** با عملگر تعیین شده است، و عدد آغازین محدوده معادل با عددی است که با عبارت **rand % 6** جمع شده است (یعنی 1). می توانیم این نتیجه را بصورت زیر تعمیم دهیم

```
عدد = مقدار شیفت + rand() % فاکتور تغییر مقیاس;
```

که "مقدار شیفت" معادل با اولین عدد موجود در بازه دلخواه اعداد صحیح متوالی بوده و "فاکتور تغییر مقیاس" معادل با پهنای دلخواه محدوده اعداد صحیح متوالی می‌باشد. تجربه نشان داده که می‌توان اعداد صحیح را از مقادیر دیگری که بصورت اعداد صحیح متوالی نیستند هم ایجاد کرد.

٦-٨ مبحث آموزشی: بازی شانس و معرفی enum

یکی از بازی‌های مورد علاقه در بسیاری از نقاط جهان، بازی بنام "*craps*" است که با طاس انجام می‌شود. حال به قوانین این بازی توجه کنید:

بازیکن دو طاس می‌اندازد و هر طاس دارای شش وجه است. این وجه‌ها متشکل از 6 یا 5، 4، 3، 2، 1 نقطه هستند. پس از رها کردن طاس‌ها، مجموع نقاط موجود بر روی هر طاس محاسبه می‌شود. اگر مجموع برابر 7 یا 11 در اولین پرتاب طاس‌ها باشد، بازیکن پرتاب کننده، برنده خواهد شد. اگر مجموع 2، 3 یا 12 در اولین پرتاب طاس‌ها باشد، بازیکن پرتاب کننده بازنده خواهد بود. اگر مجموع نقاط 10 یا 9، 8، 6، 5، 4 در اولین پرتاب طاس باشد، این مجموع تبدیل به امتیاز بازیکن خواهد شد. برای برنده شدن، بازیکن باید به پرتاب طاس‌ها ادامه دهد تا به امتیاز تعیین شده دست یابد. اگر بازیکن مقدار 7 را بعد از امتیازگیری بدست آورد، بازنده می‌شود.

شبیه‌سازی این بازی در برنامه شکل ۱۱-۶ ارائه شده است. دقت کنید که بازیکن باید دو طاس را در اولین پرتاب و تمام پرتاب‌های بعدی بکار گیرد.

```
1  // Fig. 6.11: fig06_11.cpp
2  // Craps simulation.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <cstdlib> // contains prototypes for functions srand and rand
8  using std::rand;
9  using std::srand;
10
11 #include <ctime> // contains prototype for function time
12 using std::time;
13
14 int rollDice(); // rolls dice, calculates amd displays sum
15
16 int main()
17 {
18    // enumeration with constants that represent the game status
19    enum Status { CONTINUE, WON, LOST };
20
21    int myPoint; // point if no win or loss on first roll
22    Status gameStatus; // can contain CONTINUE, WON or LOST
23
24    // randomize random number generator using current time
25    srand( time( 0 ) );
26
27    int sumOfDice = rollDice(); // first roll of the dice
28
29    // determine game status and point (if needed) based on first roll
30    switch ( sumOfDice )
31    {
32       case 7: // win with 7 on first roll
33       case 11: // win with 11 on first roll
34          gameStatus = WON;
35          break;
36       case 2: // lose with 2 on first roll
37       case 3: // lose with 3 on first roll
38       case 12: // lose with 12 on first roll
39          gameStatus = LOST;
40          break;
41       default: // did not win or lose, so remember point
42          gameStatus = CONTINUE; // game is not over
43          myPoint = sumOfDice; // remember the point
44          cout << "Point is " << myPoint << endl;
45          break; // optional at end of switch
46    } // end switch
47
48    // while game is not complete
49    while ( gameStatus == CONTINUE ) // not WON or LOST
50    {
51       sumOfDice = rollDice(); // roll dice again
52
53       // determine game status
54       if ( sumOfDice == myPoint ) // win by making point
55          gameStatus = WON;
56       else
57          if ( sumOfDice == 7 ) // lose by rolling 7 before point
58             gameStatus = LOST;
59    } // end while
60
61    // display won or lost message
62    if ( gameStatus == WON )
63       cout << "Player wins" << endl;
64    else
65       cout << "Player loses" << endl;
66
67    return 0; // indicates successful termination
68 } // end main
69
70 // roll dice, calculate sum and display results
71 int rollDice()
72 {
73    // pick random die values
74    int die1 = 1 + rand() % 6; // first die roll
75    int die2 = 1 + rand() % 6; // second die roll
```

```
76
77    int sum = die1 + die2; // compute sum of die values.
78
79    // display results of this roll
80    cout << "Player rolled " << die1 << " + " << die2
81       << " = " << sum << endl;
82    return sum; // return sum of dice
83 } // end function rollDice
```

```
Player rolled 2 + 5 = 7
Player wins
```

```
Player rolled 6 + 6 = 12
Player loses
```

```
Player rolled 3 + 3 = 6
Point is 6
Player rolled 5 + 3 = 8
Player rolled 4 + 5 = 9
Player rolled 2 + 1 = 3
Player rolled 1 + 5 = 6
Player wins
```

```
Player rolled 1 + 3 = 4
Point is 4
Player rolled 4 + 6 = 10
Player rolled 2 + 4 = 6
Player rolled 6 + 4 = 10
Player rolled 2 + 3 = 5
Player rolled 2 + 4 = 6
Player rolled 1 + 1 = 2
Player rolled 4 + 4 = 8
Player rolled 4 + 3 = 7
Player loses
```

شکل ۱۱-٦ | شبیه‌سازی Craps.

برنامه‌نویسی ایده‌ال

در نام ثابت‌های شمارشی از حروف بزرگ استفاده کنید. این کار سبب مشخص شدن این ثابت‌ها در برنامه می‌شود و به برنامه‌نویس یاد آوری می‌کند که اینها ثابت‌های شمارشی هستند و نه متغیر.

به متغیرهایی که از سوی کاربر و از نوع **status** تعریف شده‌اند فقط یکی از سه مقدار اعلان شده در نوع شمارشی را می‌توان تخصیص داد. زمانیکه بازی با برد تمام می‌شود، برنامه متغیر **gameStatus** را با **WON** تنظیم می‌کند (خطوط 34 و 55). زمانیکه برنامه با باخت تمام می‌شود، برنامه متغیر **gameStatus** را با **CONTINUE** تنظیم می‌کند (خط 42) تا نشان داده شود که تاس‌ها باید مجددا پرتاب شوند.

یک نوع شمارشی پرطرفدار دیگر در زیر آورده شده است

```
enum Months { JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NPV,
DEC};
```

که در آن نوع تعریف شده از سوی کاربر، **Month** با ثابت‌های شمارشی که نشاندهنده ماه‌های سال هستند، ایجاد شده است. اولین مقدار در نوع شمارشی فوق بطور صریح با 1 تنظیم شده است، سایر مقادیر به ترتیب هر یک، یک واحد افزایش یافته و در نتیجه مقادیر1 الی 12 تولید می‌شوند. به هر ثابت نوع شمارشی می‌توان یک عدد صحیح در تعریف شمارشی تخصیص داد، و ثابت‌های شمارشی بعد از آن با مقداری به اندازه یک واحد بیشتر از مقدار قبل از خود خواهند داشت. این حالت تا زمانیکه مجددا بصورت صریح مقداری مشخص گردد ادامه می‌یابد.

پس از اولین پرتاب، اگر بازی با برد یا باخت همراه شود، برنامه از بدنه عبارت **while** (خطوط 59-49) پرش می‌کند، چرا که **gameStatus** برابر با **CONTINUE** نیست. برنامه با عبارت **if…else** موجود در خطوط 65-62 ادامه پیدا می‌کند. اگر **gameStatus** برابر با **WON** باشد جمله "Player wins" و اگر **gameStatus** برابر با **LOST** باشد، جمله "Player loses" چاپ می‌شود.

پس از اولین پرتاب اگر بازی به پایان نرسد برنامه مجموع را در **mypoint** ذخیره می‌کند(خط 43). اجرا با عبارت **while** ادامه پیدا می‌کند، زیرا **gameStatus** برابر با **CONTINUE** است. در جریان هر تکرار **while**، برنامه مبادرت به فراخوانی **rollDice** برای تولید **sum** جدید می‌کند. اگر **sum** با **myPoint** مطابقت کند، برنامه، مقدار **gameStatus** را به **WON** تغییر داده (خط 55)، تست **while** برقرار نشده، عبارت **if…else** جمله "Player wins" را چاپ کرده و اجرا خاتمه می‌یابد. اگر **sum** معادل با 7 باشد، برنامه، مقدار **gameStatus** را به **LOST** تغییر داده (خط 58)، تست **while** برقرار شده، عبارت **if…else** جمله "Player loses" را چاپ کرده و اجرا خاتمه می‌یابد.

به کاربرد جالب انواع مکانیزم‌های کنترلی که تا بدین مرحله مورد بحث قرار داده‌ایم توجه کنید. برنامه craps از دو تابع **rollDice** و **main** به همراه عبارات **switch**، **while**، **if…else**، **if…else** تودرتو و **if** تودرتو استفاده کرده است.

برنامه‌نویسی ایده‌ال

استفاده از نوع‌های شمارشی بجای ثابت‌های صحیح، می‌تواند وضوح برنامه‌ها را افزایش داده و نگهداری آنها را بهتر سازد. می‌توانید مقدار یک ثابت شمارشی را یک بار و در اعلان نوع شمارشی مشخص نماید.

خطای برنامه‌نویسی

تخصیص معادل صحیح یک ثابت شمارشی به متغیری از نوع شمارشی، خطای کامپایل است.

خطای برنامه‌نویسی

پس از تعریف ثابت شمارشی، مبادرت به تخصیص یک مقدار دیگر به ثابت شمارشی، خطای کامپایل است.

۹-٦ کلاس‌های ذخیره‌سازی

برنامه‌هایی که تا بدین مرحله مشاهده کرده‌اید، از شناسه‌ها برای اسامی متغیرها استفاده می‌کردند. صفات متغیرها شامل نام، نوع، اندازه و مقدار است. همچنین در این فصل از شناسه‌ها بعنوان اسامی توابع تعریف شده از سوی کاربر استفاده شده است. در واقع، هر شناسه در یک برنامه دارای صفات دیگری شامل *کلاس ذخیره‌سازی*، قلمرو و *پیوند* (*linkage*) است.

زبان C++ پنج تصریح کننده کلاس ذخیره‌سازی تدارک دیده است: **auto**، **register**، **extern**، **mutable** و **static**. در این بخش کلاس‌های ذخیره‌سازی **auto**، **register**، **extern** و **static** مورد بحث قرار خواهند گرفت.

کلاس‌های ذخیره‌سازی، قلمرو و پیوند

شناسه یک کلاس ذخیره‌سازی، تعیین کننده مدت زمانی است، که در آن شناسه در حافظه وجود دارد. برخی از شناسه‌ها مدت زمان کوتاهی در حافظه وجود دارند، برخی به تناوب ایجاد و نابود می‌شوند و بقیه در کل مدت زمان اجرای برنامه در حافظه وجود دارند. در این بخش به بررسی دو کلاس ذخیره‌سازی می‌پردازیم: **static** و **automatic**.

قلمرو یک شناسه مکانی است که شناسه از آن قسمت در برنامه، می‌تواند مورد مراجعه قرار گیرد. تعدادی از شناسه‌ها می‌توانند در سرتاسر یک برنامه مورد مراجعه قرار گیرند، برخی دیگر می‌توانند فقط در قسمت‌های محدودی از یک برنامه مورد مراجعه قرار گیرند. بخش ۱۰-۶ در ارتباط با قلمرو شناسه‌ها است.

پیوند یک شناسه تعیین می‌کند که آیا شناسه فقط در فایل منبع که در آن اعلان شده است شناخته شود یا در میان فایل‌های مضاعف که کامپایل شده‌اند و سپس به یکدیگر لینک شده‌اند هم شناخته شود. کلاس ذخیره‌سازی یک شناسه در تعیین کلاس ذخیره‌سازی و پیوند آن نقش دارد.

رده‌بندی کلاس ذخیره‌سازی

می‌توان کلاس‌های ذخیره‌سازی را به دو گروه یا رده تقسیم کرد: کلاس ذخیره‌سازی اتوماتیک و کلاس ذخیره‌سازی استاتیک. از کلمات کلیدی **auto** و **register** برای اعلان متغیرهایی از کلاس ذخیره‌سازی اتوماتیک استفاده می‌شود. چنین متغیرهایی زمانی ایجاد می‌شوند که اجرای برنامه وارد بلوکی شود که آنها در آن تعریف شده‌اند، این متغیرها تا زمانی که بلوک فعال باشد، وجود خواهند داشت و زمانیکه برنامه از بلوک خارج شود، نابود خواهند شد.

متغیرهای محلی

فقط متغیرهای محلی یک تابع می‌توانند از کلاس ذخیره‌سازی اتوماتیک باشند. معمولا پارامترها و متغیرهای محلی یک تابع، از نوع کلاس ذخیره‌سازی اتوماتیک هستند. تصریح کننده کلاس ذخیره سازی

اتوماتیک بصورت صریح متغیرها را از کلاس ذخیره‌سازی اتوماتیک اعلان می کند. برای مثال، اعلان زیر بر این نکته دلالت دارد که متغیرهای **x** و **y** متغیرهای محلی از کلاس ذخیره‌سازی اتوماتیک هستند، این متغیرها فقط در نزدیکترین جفت براکت، بدنه تابعی که در آن تعریف شده اند وجود دارند:

```
auto double x, y;
```

متغیرهای محلی بصورت پیش فرض دارای کلاس ذخیره‌سازی اتوماتیک می‌باشند، از اینرو کلمه کلیدی **auto** بندرت بکار گرفته می‌شود. در مابقی متن، به متغیرهای کلاس ذخیره‌سازی اتوماتیک، فقط با نام متغیرهای اتوماتیک اشاره خواهیم کرد.

کارائی

ذخیره‌سازی اتوماتیک، وسیله‌ای برای صرفه‌جویی در مصرف حافظه است، چرا که متغیرهای کلاس ذخیره‌سازی اتوماتیک فقط زمانی در حافظه وجود دارند که بلوکی که در آن تعریف شده‌اند در حال اجرا باشد.

مهندسی نرم‌افزار

ذخیره‌سازی اتوماتیک مثالی از اصل حداقل مجوز دسترسی است، که از اصول بنیادین و خوب مهندسی نرم افزار است. بر پایه این اصل آن میزان از حق دسترسی باید به کد اعطا شود که برای انجام وظیفه تعیین شده به آن نیاز دارد و نه بیشتر. به چه دلیلی باید متغیرهایی در حافظه ذخیره و دسترس قرار دهیم که مورد نیاز نیستند؟

متغیرهای *register*

معمولا داده‌ها در نسخه زبان ماشین یک برنامه، برای انجام محاسبات و سایر پردازش‌ها به ثبات‌ها یا رجیسترها بارگذاری می‌شوند.

کارائی

*تصریح کننده کلاس ذخیره‌سازی **register** می‌تواند قبل از اعلان یک متغیر اتوماتیک قرار گیرد تا به کامپایلر اعلان کند که متغیر را در یکی از ثبات‌های سخت افزاری کامپیوتر که سرعت بسیار زیادی دارند ذخیره سازد و نه در حافظه. اگر متغیرهای پرکاربردی همانند شمارنده‌ها و مجموع‌ها در ثبات‌های سخت‌افزاری نگهداری شوند، سربار بارگذاری متناوب متغیرها از حافظه به ثبات‌ها و برگشت نتایج به حافظه مرتفع می‌شود.*

خطای برنامه‌نویسی

*استفاده از چندین کلاس ذخیره‌سازی برای یک شناسه خطای نحوی است. به یک شناسه می‌توان فقط یک کلاس ذخیره‌سازی اعمال کرد. برای مثال، اگر از **register** استفاده کنید ، دیگر **auto** را نمی توان بکار گرفت.*

کامپایلر می‌تواند اعلان‌های **register** را نادیده بگیرد. برای مثال، امکان دارد تعداد کافی از ثبات‌ها برای استفاده کامپایلر موجود نباشد. تعریف زیر پیشنهاد می‌دهد که متغیر صحیح **counter** در یکی از ثبات‌های

کامپیوتر قرار داده شود، صرفنظر از اینکه آیا کامپایلر این عمل را انجام دهد یا خیر، **counter** با 1 مقداردهی اولیه شده است :

```
register int counter = 1;
```

کلمه کلیدی **register** فقط می‌تواند با متغیرهای محلی و پارامترهای تابع بکار گرفته شود.

کارائی

*غالبا، ضرورتی به استفاده از ثبات نیست. کامپایلرهای بهینه شده امروزی قادر به شناسایی متغیرهایی هستند که متناوبا بکار گرفته می‌شوند و می‌توانند بدون اینکه نیازی باشد که برنامه‌نویس متغیری را از نوع **register**اعلان کند، تعیین می‌کنند که متغیر در ثبات قرار داده شود یا خیر.*

کلاس ذخیره‌سازی استاتیک

کلمات کلیدی **extern** و **static** شناسه‌های برای توابع و متغیرهای کلاس ذخیره‌سازی استاتیک اعلان می‌کنند. متغیرهای کلاس ذخیره‌سازی استاتیک از نقطه‌ای که برنامه شروع می‌شود، به وجود می‌آیند و طول عمرآنها تا اتمام برنامه است. حافظه یک متغیر کلاس ذخیره‌سازی استاتیک، به هنگام شروع اجرای برنامه اخذ می‌شود. چنین متغیری یک بار در زمان اعلان آن مقداردهی اولیه می‌شود. در مورد توابع، نام تابع در زمان شروع اجرای برنامه به وجود می‌آید، همانند همه توابع دیگر. با این وجود، حتی اگر متغیرها و نام توابع از زمان شروع اجرای برنامه وجود داشته باشند، این بدان معنی نیست که این شناسه‌ها می‌توانند در سرتاسر برنامه بکار گرفته شوند. کلاس ذخیره‌سازی و قلمرو (مکانی که نام می‌تواند بکار گرفته شود) مباحث جدا از یکدیگر هستند و در بخش ۱۰-۶ به این موضوع پرداخته شده است.

شناسه با کلاس ذخیره‌سازی استاتیک

دو نوع شناسه برای کلاس ذخیره‌سازی استاتیک وجود دارد، شناسه‌های خارجی (همانند اسامی متغیرهای سراسری و توابع سراسری) و متغیرهای محلی اعلان شده با تصریح کننده کلاس ذخیره‌سازی استاتیک. متغیرهای سراسری با قرار دادن اعلان‌های متغیر در خارج از تعریف تابع یا کلاس ایجاد می‌شوند. متغیرهای سراسری مقادیر خود را در کل زمان اجرای برنامه حفظ می‌کنند. متغیرهای سراسری و توابع سراسری می‌توانند از طرف هر تابعی که بعد از اعلان یا تعریف آنها در فایل منبع ظاهر می‌شود، بکار گرفته شود.

مهندسی نرم‌افزار

*اعلان یک متغیر سراسری بجای متغیر محلی، امکان رخ دادن اثرات جانبی ناخواسته دارد، زمانیکه تابعی بدون نیاز به این متغیر، به صورت تصادفی یا عمدی به تغییر مقدار آن اقدام کند. اینحالت مثال دیگری از اصل اعطا حداقل مجوزها است. بطور کلی، به جز برای منابع سراسری واقعی همانند **cin** و **cout**، از بکار بردن متغیرهای سراسری اجتناب نمائید، مگر در مواقعی که استفاده از آن سبب افزایش کارآیی برنامه گردد.*

مهندسی نرم‌افزار

متغیرهایی که فقط در یک تابع خاص بکار گرفته می‌شوند، باید بصورت متغیرهای محلی بجای متغیرهای سراسری اعلان شوند.

متغیرهای محلی اعلان شده با کلمه‌کلیدی **static** فقط در تابعی که در آن اعلان شده‌اند شناخته می‌شوند، اما، برخلاف متغیرهای اتوماتیک، متغیرهای محلی استاتیک مقادیر خود را حتی زمانیکه اجرای برنامه از تابع به فراخوان تابع برگشت داده می‌شود نیز حفظ می‌گردد. بار دیگر که تابع فراخوانی شود، متغیرهای محلی استاتیک حاوی مقادیری هستند که در آخرین اجرای تابع داشتند. عبارت زیر متغیر محلی **count** را بصورت استاتیک اعلان و با 1 مقداردهی اولیه کرده است:

```
static int count = 1;
```

تمام متغیرهای عددی از کلاس ذخیره‌سازی استاتیک با صفر مقدار دهی اولیه می‌شوند، اگر بصورت صریح از طرف برنامه‌نویس مقداردهی اولیه نشده باشند، اما مقداردهی اولیه تمامی متغیرها بصورت صریح کار چندان مناسبی نیست.

تصریح کننده‌های کلاس ذخیره‌سازی **extern** و **static** زمانیکه بصورت صریح با شناسه‌های خارجی همانند اسامی متغیرهای سراسری و توابع سراسری بکار گرفته می‌شوند، معنی خاصی پیدا می‌کنند.

۱۰-٦ قوانین قلمرو

به بخشی از برنامه که یک شناسه در آن می‌تواند بکار گرفته شود، قلمرو آن شناسه گفته می‌شود. برای مثال، هنگامی که یک متغیر محلی در یک بلوک اعلان می‌شود، آن متغیر فقط می‌تواند در آن بلوک و در بلوک‌هایی که بصورت تو در تو دورن آن بلوک قرار دارند، مورد مراجعه قرار گیرد. در این بخش به بررسی چهار قلمرو موجود برای یک شناسه می‌پردازیم: قلمرو تابع، قلمرو فایل، قلمرو بلوکی و قلمرو نمونه اولیه تابع.

شناسه اعلان شده در خارج از تابع یا کلاس دارای قلمرو فایل است. چنین شناسه‌ای از مکانی که در آن اعلان شده تا انتهای فایل در تمامی توابع شناخته می‌شود. متغیرهای سراسری، تعریف تابع و نمونه‌های اولیه تابع قرار گرفته در خارج از تابع، همگی دارای قلمرو فایل می‌باشند.

برچسب‌ها (شناسه‌هایی که پس از آنها یک کولن قرار داده می‌شود همانند :start) تنها شناسه‌های با قلمرو تابع هستند. برچسب‌ها می‌توانند در هر جای تابع که در آن ظاهر می‌شوند بکار گرفته شوند، اما در خارج از بدنه تابع نمی‌توانند مورد مراجعه قرار گیرند. از برچسب‌ها در عبارات goto استفاده می‌شود. برچسب‌ها پیاده‌سازی کننده جزئیاتی هستند که توابع آنها را از یکدیگر پنهان نگه می‌دارند.

شناسه‌هایی که درون یک بلوک اعلان شده‌اند، دارای قلمرو بلوکی هستند. قلمرو بلوکی از مکان اعلان شناسه آغاز شده و در مکانی که براکت بسته (}) بلوکی که شناسه در آن اعلان شده قرار دارد، خاتمه

می‌پذیرد. متغیرهای محلی دارای قلمرو بلوکی هستند، همانند پارامترهای تابع که متغیرهای محلی تابع نیز می‌باشند. هر بلوک می‌تواند حاوی اعلان متغیرها باشد. زمانیکه بلوک‌ها بصورت تودرتو هستند و شناسه‌ای همنام با شناسه موجود در بلوک داخلی وجود داشته باشد، شناسه در بلوک خارجی تا زمانی که بلوک داخلی خاتمه یابد، پنهان می‌شود. در زمان اجرای بلوک داخلی، بلوک داخلی مقدار شناسه محلی خود را می‌بیند و به مقدار شناسه همنام خود در بلوک در برگیرنده خود توجهی ندارد. متغیرهای محلی استاتیک هم دارای قلمرو بلوکی هستند، حتی اگر از زمان شروع برنامه وجود داشته باشند. مدت زمان ذخیره‌سازی تاثیری در قلمرو یک شناسه ندارد.

تنها شناسه‌هایی که دارای قلمرو نمونه‌های اولیه تابع می‌باشند، شناسه‌های هستند که در لیست پارامتری نمونه اولیه تابع بکار رفته‌اند. همانطوری که قبلا هم گفته شد، در نمونه‌های اولیه تابع در لیست پارامتری نیازی به حضور نام نیست و فقط نوع آنها مورد نیاز است. اسامی بکار رفته در لیست پارامتری یک نمونه اولیه تابع از طرف کامپایلر نادیده گرفته می‌شوند. شناسه‌هایی که در نمونه اولیه یک تابع بکار برده می‌شوند، می‌توانند در هر کجای برنامه مورد استفاده مجدد قرار گیرند، بدون اینکه ابهامی بوجود آورند. در یک نمونه اولیه منفرد، یک شناسه خاص فقط می‌تواند یکبار بکار برده شود.

خطای برنامه‌نویسی

استفاده از نام مشابه برای یک شناسه در بلوک داخلی و بلوک خارجی، زمانیکه برنامه‌نویس می‌خواهد به شناسه موجود در بلوک خارجی دسترسی پیدا کند، معمولا خطای منطقی بدنبال دارد.

برنامه‌نویسی ایده‌ال

از نامگذاری که سبب می‌شود اسامی در قلمرو خارجی پنهان شوند اجتناب کنید.

برنامه شکل ۱۲-۶ به توصیف مباحث قلمرو در متغیرهای سراسری، متغیرهای محلی اتوماتیک و متغیرهای محلی استاتیک پرداخته است.

```
1  // Fig. 6.12: fig06_12.cpp
2  // A scoping example.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  void useLocal( void ); // function prototype
8  void useStaticLocal( void ); // function prototype
9  void useGlobal( void ); // function prototype
10
11 int x = 1; // global variable
12
13 int main()
14 {
15    int x = 5; // local variable to main
16
17    cout << "local x in main's outer scope is " << x << endl;
18
19    { // start new scope
20       int x = 7; // hides x in outer scope
21
22       cout << "local x in main's inner scope is " << x << endl;
```

```
23    } // end new scope
24
25    cout << "local x in main's outer scope is " << x << endl;
26
27    useLocal(); // useLocal has local x
28    useStaticLocal(); // useStaticLocal has static local x
29    useGlobal(); // useGlobal uses global x
30    useLocal(); // useLocal reinitializes its local x
31    useStaticLocal(); // static local x retains its prior value
32    useGlobal(); // global x also retains its value
33
34    cout << "\nlocal x in main is " << x << endl;
35    return 0; // indicates successful termination
36 } // end main
37
38 // useLocal reinitializes local variable x during each call
39 void useLocal( void )
40 {
41    int x = 25; // initialized each time useLocal is called.
42
43    cout << "\nlocal x is " << x << " on entering useLocal" << endl;
44    x++;
45    cout << "local x is " << x << " on exiting useLocal" << endl;
46 } // end function useLocal
47
48 // useStaticLocal initializes static local variable x only the
49 // first time the function is called; value of x is saved
50 // between calls to this function
51 void useStaticLocal( void )
52 {
53    static int x = 50; // initialized first time useStaticLocal is called
54
55    cout << "\nlocal static x is " << x << " on entering useStaticLocal"
56       << endl;
57    x++;
58    cout << "local static x is " << x << " on exiting useStaticLocal"
59       << endl;
60 } // end function useStaticLocal
61
62 // useGlobal modifies global variable x during each call
63 void useGlobal( void )
64 {
65    cout << "\nglobal x is " << x << " on entering useGlobal" << endl;
66    x *= 10;
67    cout << "global x is " << x << " on exiting useGlobal" << endl;
68 } // end function useGlobal
```

```
local x in main's outer scope is 5
local x in main's inner scope is 7
local x in main's outer scope is 5

local x is 25 on entering useLocal
local x is 26 on exiting useLocal

local static x is 50 on entering useStaticLocal
local static x is 51 on exiting useStaticLocal

global x is 1 on entering useGlobal
global x is 10 on exiting useGlobal

local x is 25 on entering useLocal
local x is 26 on exiting useLocal

local static x is 50 on entering useStaticLocal
local static x is 52 on exiting useStaticLocal

global x is 10 on entering useGlobal
global x is 100 on exiting useGlobal

local x in main is 5
```

اعلان و با 5 مقداردهی اولیه کردنه است. خط 17 برای نشان دادن این موضوع که **x** سراسری در **main** پنهان است، این متغیر را چاپ کرده است. سپس، خطوط 23-19 یک بلوک جدید در **main** تعریف می‌کنند که در آن یک متغیر محلی بنام **x** اعلان و با 7 مقداردهی اولیه شده است (خط 20). خط 22 این متغیر را چاپ می‌کند تا نشان دهد که **x** در بلوک خارجی **main** پنهان است. زمانیکه بلوک به پایان می‌رسد، متغیر **x** با مقدار 7 بصورت اتوماتیک نابود می‌شود. سپس، خط 25 متغیر محلی **x** موجود در بلوک خارجی **main** را چاپ می‌کند تا نشان داده شود که این متغیر دیگر پنهان شده نیست.

برای توصیف قلمروهای دیگر، برنامه سه تابع تعریف کرده است، آرگومانی دریافت نکرده و چیزی برگشت نمی‌دهند. تابع **useLocal** (خطوط **46-39**) متغیر اتوماتیک **x** را اعلان (خط 41) و با 25 مقداردهی اولیه کرده است. زمانیکه برنامه **useLocal** را فراخوانی می‌کند، تابع مقدار متغیر را چاپ کرده، یک واحد آنرا افزایش داده و قبل از آنکه تابع، کنترل برنامه را فراخوان خود برگشت دهد، مجددا آنرا چاپ می‌کند. هر بارکه برنامه این تابع را فراخوانی می‌کند، تابع مجددا متغیر اتوماتیک **x** را ایجاد و با 25 مقداردهی اولیه می‌کند.

تابع **useStaticLocal** (خطوط 60-51) متغیر استاتیک **x** را و آن را با 50 مقداردهی اولیه کرده است. متغیرهای محلی اعلان شده بصورت استاتیک مقادیر خود را حتی هنگامی که خارج از قلمرو هستند (یعنی در تابعی که در آن اعلان شده و آن تابع در حال اجرا نباشد) حفظ می‌کنند. زمانیکه برنامه مبادرت به فراخوانی **useStaticLocal** می‌کند، تابع مقدار **x** را چاپ کرده، آنرا یک واحد افزایش داده و قبل از آنکه تابع مبادرت به برگرداندن کنترل برنامه به تابع فراخوان خود نماید، مجددا آنر چاپ می‌کند. در فراخوانی بعدی این تابع، متغیر محلی استاتیک **x** حاوی مقدار 51 است. مقداردهی بکار رفته در خط 53 فقط یکبار رخ می‌دهد، اولین بار که **useStaticLocal** فراخوانی می‌شود.

تابع **useGlobal** (خطوط 68-63) هیچ متغیری را اعلان نکرده است. بنابر این، هنگامی که به متغیر **x** مراجعه می‌کند، **x** سراسری (**main** قبلی) بکار گرفته می‌شود. زمانیکه برنامه تابع **useGlobal** را فراخوانی می‌کند، تابع متغیر سراسری **x** را چاپ کرده، آنرا در 10 ضرب و قبل از آنکه تابع مبادرت به برگرداندن کنترل برنامه به فراخوان خود نماید، مجددا آن را چاپ می‌کند. بار دیگر که برنامه، تابع **useGlobal** را فراخوانی می‌کند، متغیر سراسری حاوی مقدار تغییر یافته، 10 است. پس از آنکه هر یک از توابع **useLocal**، **useStaticLocal** و **useGlobal** دو بار اجرا شدند، برنامه مقدار متغیر محلی **x** موجود در **main** را مجددا چاپ می‌کند تا نشان دهد که هیچ کدامیک از فراخوانی‌های تابع مقدار **x** موجود در **main** را تغییر نداده‌اند، چرا که تمام توابع به متغیرهای موجود در قلمروهای دیگر مراجعه دارند.

۱۱-٦ عملکرد پشته فراخوانی و ثبت فعالیت‌ها

برای درک نحوه فراخوانی تابع در C++، ابتدا نیاز است به بررسی ساختمان داده (یعنی کلکسیونی از ایتم‌های داده مرتبط با هم) بنام پشته بپردازیم. می‌توانید پشته را همانند تعدادی بشقاب که روی هم قرار گرفته‌اند، در نظر بگیرید. هنگامی که بشقابی بر روی سایر بشقاب‌ها قرار داده می‌شود، معمولا در بالای بشقاب‌ها جای داده می‌شود (به این عمل گذاشتن یا ***push*** کردن بشقاب در پشته گفته می‌شود). به طور مشابه، هنگامی که یک بشقاب از روی بشقاب‌ها برداشته می‌شود، معمولا از بالای بشقاب‌ها برداشته می‌شود (به این عمل برداشتن یا ***pop*** کردن بشقاب از پشته گفته می‌شود). پشته‌ها بعنوان ساختمان‌های داده LIFO (last-in, last-out) شناخته می‌شوند، به این معنی که آخرین ایتمی که در پشته گذاشته می‌شود، اولین ایتمی است که از پشته برداشته می‌شود.

یکی از مهمترین مکانیزم‌های که باید از طرف دانشجویان کامپیوتر درک شود، پشته فراخوانی تابع است (بعنوان پشته اجرای برنامه هم نامیده می‌شود). این ساختمان داده که در پشت صحنه کار می‌کند، از مکانیزم فراخوانی/برگشت تابع پشتیبانی می‌کند. همچنین از ایجاد، نگهداری و نابود کردن متغیرهای اتوماتیک توابع فراخوانی شده پشتیبانی می‌نماید. رفتار LIFO پشته‌ها را با مثال بشقاب‌ها توضیح دادیم. همانطوری که در برنامه‌های شکل ۱۴-۶ الی ۱۶-۶ خواهید دید، این رفتار LIFO دقیقا همان کاری است که تابع ، در زمان بازگشت به تابع فراخوان خود انجام می‌دهد.

زمانیکه تابعی فراخوانی می‌شود، امکان دارد قبل از آنکه برگشت یابد، توابع دیگری را فراخوانی کند، به همین ترتیب امکان دارد این توابع هم، قبل از برگشت به تابع فراخوان خود، توابع دیگری را فراخوانی کنند. سرانجام هر تابعی باید کنترل را به تابع فراخوان خود باز گرداند. از اینرو، باید به روشی، آدرس‌های بازگشت را که هر تابع برای بازگرداندن کنترل به تابع فراخوان خود به آنها نیاز دارد ردگیری و نگهداری کنیم. پشته فراخوان تابع یک ساختمان داده عالی برای رسیدگی به این اطلاعات است. هر زمانیکه تابعی مبادرت به فراخوانی تابع دیگر می‌کند، یک ورودی در پشته ثبت می‌شود یا به عبارتی به پشته push می‌گردد. این ورودی، یک *فریم پشته* (*stack frame*) یا *ثبت فعالیت‌ها* نامیده می‌شود، و حاوی آدرس بازگشت است که تابع فراخوانده شده برای برگشت به تابع فراخوان خود به آن نیاز دارد. همچنین فریم پشته حاوی برخی از اطلاعات دیگر هم است که به زودی در مورد آنها صحبت خواهیم کرد. اگر تابع فراخوانده شده، بجای فراخوانی تابع دیگری قبل از بازگشت، به محل فراخوانی خود برگشت یابد، فریم پشته فراخوانی تابع pop شده، و کنترل به آدرس بازگشت، در فریم پشته pop شده انتقال داده می‌شود.

قابلیت پشته فراخوان در این است که هر تابع فراخوانی شده همیشه اطلاعات مورد نیاز خود برای برگشت به فراخوان خود را در بالای پشته فراخوان پیدا می‌کند. و اگر تابعی، تابع دیگری را فراخوانی کند، به ساده‌گی یک فریم پشته برای تابع جدیدا فراخوانی شده، در پشته فراخوان push می‌شود. از اینرو، هم

اکنون آدرس بازگشت مورد نیاز برای برگشت تابع جدیدا فراخوانی شده به فراخوان خود در بالای پشته قرار دارد.

فریم‌های پشته مسئولیت مهم دیگری هم بر عهده دارند. اکثر توابع دارای متغیرهای اتوماتیک همانند پارامترها و متغیرهای محلی که تابع اعلان می‌کند، هستند. متغیرهای اتوماتیک باید در زمان اجرای یک تابع وجود داشته باشند. اگر تابع مبادرت به فراخوانی توابع دیگری کند، این متغیرها باید فعال نگه داشته شوند. ام زمانیکه تابع فراخوانده شده به فراخوان خود باز می‌گردد، نیاز است تا متغیرهای اتوماتیک تابع فراخوانده شده از بین بروند. فریم پشته تابع فراخوانده شده، مکانی عالی برای رزرو حافظه برای متغیرهای اتوماتیک تابع فراخوانده شده است. این فریم پشته تا زمانیکه تابع فراخوانده شده فعال است وجود خواهد داشت. زمانیکه تابع فراخوانده شده برگشت پیدا می‌کند و دیگر نیازی به متغیرهای اتوماتیک محلی خود ندارد، فریم پشته آن از پشته pop شده، و دیگر این متغیرهای اتوماتیک محلی در برنامه شناخته نمی‌شوند.

البته، مقدار حافظه در کامپیوتر با محدودیت همراه است، از اینرو فقط مقدار مشخصی از حافظه می‌تواند برای ذخیره ثبت فعالیت‌ها در پشته فراخوان تابع بکار گرفته شود. اگر تعداد فراخوانی‌های توابع بیش از مقدار و توان حافظه در نظر گرفته شده به این منظور باشد، با خطای بنام *سرریز پشته* (*stack overflow*) مواجه خواهید شد.

پشته فراخوانی تابع در عمل

بسیار خوب، همانطوری که مشاهده کردیم، پشته فراخوانی و ثبت فعالیت‌ها، از مکانیزم فراخوانی/برگشت دادن تابع، ایجاد و نابود کردن متغیرهای اتوماتیک پشتیبانی می‌کنند. اجازه دهید به بررسی پشتیبانی پشته فراخوانی از عملیات تابع **square** که توسط **main** فراخوانی می‌شود بپردازیم (خطوط 17-11 از شکل۱۳-۶). ابتدا سیستم عامل تابع **main** را فراخوانی می‌کند، اینکار سبب push شدن یک رکورد فعالیت در پشته می‌شود (در شکل ۱۴-۶ نشان داده شده است). رکورد فعالیت به **main** نحوه برگشت به سیستم عامل را بیان کرده (یعنی انتقال به آدرس برگشت **R1**) و حاوی فضا برای متغیر اتوماتیک **main** است (یعنی **a** که با 10 مقداردهی اولیه شده است).

```
1  // Fig. 6.13: fig06_13.cpp
2  // square function used to demonstrate the function
3  // call stack and activation records.
4  #include <iostream>
5  using std::cin;
6  using std::cout;
7  using std::endl;
8
9  int square( int ); // prototype for function square
10
11 int main()
12 {
13    int a = 10; // value to square (local automatic variable in main).
14
15    cout << a << " squared: " << square( a ) << endl;// display a squared
16    return 0; // indicate successful termination
```

```
17 } // end main
18
19 // returns the square of an integer
20 int square( int x ) // x is a local variable
21 {
22    return x * x; // calculate square and return result
23 } // end function square
```

```
10 squared: 100
```

شکل ۱۳-٦ | استفاده از تابع square برای توصیف عملکرد پشته فراخوان تابع و رکورد فعالیت.

شکل ١٤-٦ | پشته فراخوانی تابع پس از اینکه سیستم‌عامل تابع main را برای اجرای برنامه فراخوانی کرده است.

حال تابع **main**، تابع **square** در خط 15 از شکل ۱۳-۶ را قبل از برگشت به سیستم عامل فراخوانی می‌کند. این عمل سبب می‌شود تا یک فریم پشته برای **square** (خطوط 23-20) در پشته فراخوانی تابع push شود (شکل ۱۵-۶). این فریم پشته، حاوی آدرس برگشتی است که **square** برای برگشت به **main** (یعنی **R2**) و حافظه برای متغیر اتوماتیک **x** به آن نیاز دارد.

پس از اینکه **square** مربع آرگومان خود را بدست آورد، نیاز به برگشت به **main** دارد و دیگر نیازی به حافظه برای متغیر اتوماتیک خود یعنی **x** ندارد. از اینرو pop بر روی پشته اعمال می‌شود، امکان برگشت **square** به **main** (یعنی **R2**) فراهم شده و متغیر اتوماتیک **square** از بین می‌رود. شکل ۱۶-۶ پشته فراخوانی تابع پس از **pop** شدن رکورد فعالیت **square** است.

اکنون تابع **main** مبادرت بنمایش نتیجه فراخوانی **square** می‌کند (خط 15)، سپس عبارت **return** را اجرا می‌نماید (خط 16). این عمل سبب می‌شود تا رکورد فعالیت **main** از پشته pop شود. با اینکار آدرس مورد نیاز برای بازگشت به سیستم عامل (**R1** در شکل ۱۴-۶) به **main** داده شده و سبب می‌شود حافظه متغیر اتوماتیک **main** (یعنی **a**) در دسترس نباشد.

حال مشاهده کردید که چگونه ساختمان داده پشته مبادرت به پیاده‌سازی مکانیزمی کلیدی می‌کند که از اجرای برنامه‌ها پشتیبانی می‌نماید. ساختمان‌های داده کاربردهای مهمی در علم کامپیوتر دارند. در فصل بیست‌ویکم در ارتباط با پشته‌ها، صف‌ها، لیست‌ها، درخت‌ها و سایر ساختمان‌های داده صحبت خواهیم کرد.

شکل ١٥-٦ | پشته فراخوانی تابع پس از اینکه main تابع square را برای انجام محاسبه فراخوانی کرده است.

شکل ١٦-٦ | پشته فراخوانی تابع پس از اینکه تابع square به main برگشت داده شده است.

٦-١٢ توابع با لیست پارامتری تهی

در زبان C++، یک لیست پارامتری تهی با نوشتن **void** یا خالی گذاشتن پارانتزها مشخص می‌شود. عبارت زیر

```
void print();
```

تصریح می‌کند که تابع **print** آرگومانی دریافت نمی‌کند و مقداری را هم برگشت نمی‌دهد. برنامه شکل ۱۷-۶ هر دو روش اعلان و استفاده از توابع با لیست‌های پارامتری تهی را نشان می دهد.

قابلیت حمل

مفهوم لیست پارامتری تهی تابع در C++ بطور قابل توجهی متفاوت از C است. در زبان C، لیست پارامتری تهی به معنی است که بررسی کلیه آرگومان‌ها غیر فعال است (فراخوانی تابع می‌تواند هر آرگومانی را ارسال کند). در C++، بدین معنی است که تابع بطور صریح هیچ آرگومانی دریافت نمی‌کند. بنابر این، برنامه‌های C که از ویژگی استفاده می‌کنند ممکن است در هنگام کامپایل شدن در C++ خطاهای کامپایل تولید کنند.

```
1  // Fig. 6.17: fig06_17.cpp
2  // Functions that take no arguments.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  void function1(); // function that takes no arguments
8  void function2( void ); // function that takes no arguments
9
10 int main()
11 {
12    function1(); // call function1 with no arguments
13    function2(); // call function2 with no arguments
14    return 0; // indicates successful termination
15 } // end main
16
17 // function1 uses an empty parameter list to specify that
18 // the function receives no arguments
19 void function1()
20 {
21    cout << "function1 takes no arguments" << endl;
22 } // end function1
23
24 // function2 uses a void parameter list to specify that
25 // the function receives no arguments
26 void function2( void )
27 {
28    cout << "function2 also takes no arguments" << endl;
29 } // end function2
```

```
function1 takes no arguments
function2 also takes no arguments
```

شکل ۱۷-٦ | توابعی که آرگومان دریافت نمی‌کنند.

۱۳-٦ توابع inline

پیاده‌سازی یک برنامه به صورت مجموعه‌ای از توابع از نظر مهندسی نرم‌افزار کار مناسبی است، اما فراخوانی‌های تابع، سربارگذاری زمان اجرا بدنبال دارد. C++ برای کمک به کاهش سربارگذاری فراخوانی تابع، توابع **inline** را تدارک دیده است، به ویژه برای توابع کوچک. قرار دادن توصیف کننده **inline** قبل از نوع بازگشتی تابع در تعریف تابع، به کامپایلر توصیه می‌کند در محل استفاده از تابع (در زمان مناسب) یک کپی از کد تابع ایجاد و از فراخوانی تابع ممانعت کند. مشکل اینجاست که کپی‌های مضاعف از کد تابع در برنامه وارد می‌شود و اغلب سبب بزرگتر شدن برنامه می‌گردند، بجای اینکه یک

کپی منفرد از تابع که در هر بار فراخوانی تابع کنترل به آن ارسال شود. کامپایلر می‌تواند توصیفی کننده **inline** را نادیده گرفته و عموما نیز این کار را برای تمامی توابع به جز توابع کوچک انجام می‌دهد.

مهندسی نرم‌افزار

*هر تغییری در یک تابع **inline** مستلزم کامپایل مجدد تمامی سرویس‌گیرنده‌های تابع است. اینکار می‌تواند در توسعه و نگهداری برخی از برنامه‌ها قابل توجه باشد.*

برنامه‌نویسی ایده‌ال

*باید توصیف کننده **inline** فقط با توابع کوچک و پرکاربرد بکار گرفته شود.*

کارائی

*استفاده از توابع **inline** می‌تواند زمان اجرا را کاهش دهد، اما می‌تواند سایز برنامه را هم افزایش دهد.*

برنامه شکل ۱۸-۶ از یک تابع **inline** بنام **cube** (خطوط 14-11) برای محاسبه حجم مکعبی با ضلع **side** استفاده می‌کند. کلمه‌کلیدی **const** در لیست پارامتری تابع **cube** (خط 11) به کامپایلر اعلان می‌کند که تابع مبادرت به تغییر متغیر **side** نمی‌کند. با اینکار تضمین می‌شود که مقدار **side** در هنگام انجام محاسبه از طرف تابع تغییر داده نخواهد شد. جزئیات کلمه‌کلیدی **const** در فصل هفتم، فصل هشتم و فصل دهم توضیح داده شده است. توجه کنید که تعریف کامل تابع **cube** قبل از استفاده از آن در برنامه ظاهر شده است. انجام اینکار ضروری است، چراکه با انجام اینکار کامپایلر می‌داند که چگونه فراخوانی تابع **cube** را به کد **inline** آن گسترش دهد. به همین دلیل، معمولا توابع **inline** با قابلیت استفاده مجدد در فایل‌های سرآیند قرار داده می‌شوند ، از اینرو است که تعاریف آنها می‌توانند در هر فایل منبع که از آنها استفاده می‌کند، شامل گردد.

مهندسی نرم‌افزار

*باید توصیف کننده **const** را برای پیاده کردن اصل حداقل حق مجوز دسترسی بکار گرفت. استفاده از این اصل در توسعه نرم‌افزار می‌تواند زمان خطایابی و تاثیرات جانبی را به حداقل رسانده و تغییر و نگهداری از برنامه را آسان‌تر کند.*

```
1  // Fig. 6.18: fig06_18.cpp
2  // Using an inline function to calculate the volume of a cube.
3  #include <iostream>
4  using std::cout;
5  using std::cin;
6  using std::endl;
7
8  // Definition of inline function cube. Definition of function appears
9  // before function is called, so a function prototype is not required.
10 // First line of function definition acts as the prototype.
11 inline double cube( const double side )
12 {
13    return side * side * side; // calculate cube
14 } // end function cube
15
16 int main()
17 {
18    double sideValue; // stores value entered by user
19    cout << "Enter the side length of your cube: ";
```

```
20    cin >> sideValue; // read value from user
21
22    // calculate cube of sideValue and display result
23    cout << "Volume of cube with side "
24       << sideValue << " is " << cube( sideValue ) << endl;
25    return 0; // indicates successful termination
26 } // end main
```

```
Enter the side length of your cube: 3.5
Volume of cube with side 3.5 is 42.875
```

شکل ۱۸-٦ | تابع inline که مبادرت به محاسبه حجم یک مکعب می‌کند.

١٤-٦ مراجعه و پارامترهای مراجعه

دو روش برای ارسال آرگومان به توابع در بسیاری از زبان‌های برنامه‌نویسی وجود دارد که عبارتند از "*ارسال با مقدار*" و "*ارسال با مراجعه*". هنگامی که یک آرگومان به روش مقدار ارسال می‌شود، یک کپی از مقدار آرگومان تهیه و به تابع فراخوانده شده ارسال می‌گردد (در پشته فراخوانی تابع). هر گونه تغییر در کپی، تاثیری در مقدار اصلی متغیر در فراخوان اعمال نمی‌کند. اینکار از تاثیرات جانبی و تصادفی که تا حد زیادی در توسعه سیستم‌های نرم‌فزاری صحیح و اطمینان بالا مشکل بوجود می‌آورند، جلوگیری می‌کند. تمامی آرگومانی‌های ارسالی تابدین مرحله از این فصل به روش ارسال با مقدار بودند.

کارائی

یکی از معایب ارسال به روش مقدار این است که اگر یک ایتم داده بزرگ ارسال گردد، کپی کردن آن داده می‌تواند مقدار قابل توجهی از زمان اجرا و فضای حافظه تلف کند.

پارامتر های مراجعه

این بخش به معرفی "*پارامترهای مراجعه یا ارجاعی*" می‌پردازد، یکی از دو روشی که C++ برای ارسال با مراجعه تدارک دیده است. در روش ارسال با مراجعه، فراخوان به تابع فراخوانی شده امکان دسترسی مستقیم به داده‌های خود و اصلاح آن داده‌ها را می‌دهد.

کارائی

روش ارسال با مراجعه در افزایش کارآیی موثر است، چرا که نیاز به سربارگذاری از کپی کردن میزان زیادی از داده‌ها در روش ارسال با مقدار را از بین می‌برد.

مهندسی نرم‌افزار

ارسال با مراجعه می‌تواند در کاهش امنیت تاثیر گذار باشد، چرا که تابع فراخوان می‌تواند داده‌های فراخوان خود را معیوب کند.

بعدا، نشان خواهیم داد که چگونه می‌توان از کارآیی ارسال به روش مراجعه استفاده کرده و با توجه به اصول مهندسی نرم‌افزار، از داده‌های فراخوان محافظت کرد.

یک پارامتر ارجاعی یک نام مستعار برای آرگومان متناظر خود در فراخوانی تابع است. برای نشان دادن اینکه پارامتر تابع به روش مراجعه ارسال می‌شود، کافیست پس از نوع پارامتر در نمونه اولیه تابع یک &

قرار داده شود، از همین روش به هنگام لیست کردن نوع پارامتر در سرآیند تابع هم استفاده کنید. برای مثال، اعلان زیر در سرآیند یک تابع

```
int &count
```

زمانیکه از راست به چپ خوانده شود به معنی "**count** یک مراجعه به یک **int** است" خواهد بود. در فراخوانی تابع، کافی است متغیر را با نام ذکر کنید تا به روش مراجعه ارسال شود. سپس، ذکر متغیر توسط نام پارامتر آن در بدنه تابع فراخوانده شده، در واقع یک مراجعه به متغیر اصلی در تابع فراخوان است، و متغیر اصلی می‌تواند مستقیما از طرف تابع فراخوانده شده تغییر داده شود. باز هم، باید نمونه اولیه و سرآیند تابع با هم موافق باشند.

ارسال آرگومان با مقدار و با مراجعه

برنامه شکل ۱۹-۶ به مقایسه روش ارسال با مقدار و ارسال با مراجعه با پارامترهای ارجاعی پرداخته است. سبک آرگومان‌ها در فراخوانی توابع **squareByValue** و **squareByReference** یکسان است، هر دو متغیر با نام خود در فراخوانی‌ها مشخص شده‌اند. بدون بررسی نمونه‌های اولیه تابع یا تعاریف تابع، نمی‌توان بر اساس فراخوانی‌ها تعیین کرد که تابع می‌تواند در آرگومان‌های خود تغییر بوجود آورد. بدلیل اینکه نمونه‌های اولیه تابع اجباری هستند، کامپایلر مشکلی در رفع این ابهام ندارد.

```
1  // Fig. 6.19: fig06_19.cpp
2  // Comparing pass-by-value and pass-by-reference with references.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int squareByValue( int ); // function prototype (value pass)
8  void squareByReference( int & ); // function prototype (reference pass)
9
10 int main()
11 {
12    int x = 2; // value to square using squareByValue
13    int z = 4; // value to square using squareByReference
14
15    // demonstrate squareByValue
16    cout << "x = " << x << " before squareByValue\n";
17    cout << "Value returned by squareByValue: "
18       << squareByValue( x ) << endl;
19    cout << "x = " << x << " after squareByValue\n" << endl;
20
21    // demonstrate squareByReference
22    cout << "z = " << z << " before squareByReference" << endl;
23    squareByReference( z );
24    cout << "z = " << z << " after squareByReference" << endl;
25    return 0; // indicates successful termination
26 } // end main
27
28 // squareByValue multiplies number by itself, stores the
29 // result in number and returns the new value of number
30 int squareByValue( int number )
31 {
32    return number *= number; // caller's argument not modified
33 } // end function squareByValue
34
35 // squareByReference multiplies numberRef by itself and stores the result
36 // in the variable to which numberRef refers in function main
37 void squareByReference( int &numberRef )
```

```
38 {
39    numberRef *= numberRef; // caller's argument modified
40 } // end function squareByReference
```

```
x = 2 before squareByValue
Value returend by squareByValue: 4
x = 2 after squareByValue

z = 4 before squareByReference
z = 16 after squareByReference
```

شکل ۱۹-٦ | ارسال آرگومان به روش مقدار و مراجعه.

خطای برنامه‌نویسی

بدلیل اینکه پارامترهای ارجاعی فقط با نام در بدنه تابع فراخوانی شده ذکر می‌شوند، ممکن است برنامه‌نویس سهوا با پارامترهای ارجاعی بعنوان پارامترهای ارسالی با مقدار رفتار کند. اگر کپی‌های اصلی از متغیرها توسط تابع تغییر داده شوند، اینکار می‌تواند اثرات جانبی غیرمنتظره‌ای بوجود آورد.

در فصل هشتم در ارتباط با اشاره‌گرها صحبت خواهیم کرد، اشاره‌گرها یک روش جایگزین برای ارسال با مراجعه هستند که در آن سبک فراخوانی، به وضوح دلالت بر ارسال با مراجعه دارد.

کارائی

برای ارسال شی‌های بزرگ، از پارامتر ارجاعی ثابت برای شبیه‌سازی ظاهر و امنیت ارسال با مقدار استفاده کرده و از سربارگذاری ارسال یک کپی از شیء بزرگ اجتناب کنید.

مهندسی نرم‌افزار

برخی از برنامه‌نویسان زحمت اعلان پارامترهای ارسالی با مقدار را به عنوان ثابت را، حتی در صورتی که تابع فراخوانی شده نیازی به تغییر در آرگومان‌های ارسالی نداشته باشد، به خود نمی‌دهند. کلمه کلیدی ***const*** *در این زمینه فقط می‌تواند از یک کپی از آرگومان اصلی محافظت کند، نه خود آرگومان اصلی، که در زمان ارسال به روش مقدار در مقابل تغییرات توسط تابع فراخوان ایمن است.*

برای مشخص کردن یک مراجعه به یک ثابت، توصیف کننده **const** را قبل از مشخص کننده نوع در اعلان پارامتر قرار دهید.

به خط 37 از شکل ۱۹-۶ و به مکان & در لیست پارامتری تابع **squareByReference** توجه کنید. برخی از برنامه‌نویسان C++ ترجیح می‌دهند آنرا بصورت **int& numberRef** بنویسند.

مهندسی نرم‌افزار

به منظورافزایش وضوح و کارآیی، بسیاری از برنامه‌نویسان C++ ترجیح می‌دهند آرگومان‌های تغییرپذیر را با استفاده از اشاره‌گر، آرگومان‌های کوچک غیرقابل تغییر به را روش مقدار و آرگومان‌های بزرگ غیرقابل تغییر را به روش مراجعه به ثابت‌ها، به توابع ارسال کنند.

مراجعه‌ها بعنوان اسامی مستعار در درون یک تابع

مراجعه‌ها همچنین می‌توانند بعنوان اسامی مستعار برای متغیرهای دیگر در درون یک تابع بکار برده شوند (اگرچه آنها معمولا در توابعی به شکلی که در شکل ۱۹-۶ نشان داده شده بکار گرفته می‌شوند). برای مثال، کد

```
int count = 1; // declare integer variable count
int &cRef = count; // create cRef as an alias for count
cRef++; // increment count (using its alias cRef)
```

متغیر **count** را با استفاده از نام مستعار **cRef** یک واحد افزایش می‌دهد. متغیرهای ارجاعی باید در اعلان‌های خود مقداردهی اولیه شوند (شکل ۲۰-۶ و شکل ۲۱-۶) و نمی‌توانند مجددا بعنوان اسامی مستعار به متغیرهای دیگر تخصیص داده شوند. زمانیکه یک مراجعه بعنوان نام مستعار برای متغیر دیگری اعلان شود، کلیه عملیات‌های انجام شده بر روی نام مستعار در واقع بر روی متغیر اصلی انجام می‌شوند. در حقیقت نام مستعار، نام دیگری برای متغیر اصلی به شمار می‌آید. گرفتن آدرس یک مراجعه و مقایسه مراجعه‌ها سبب رخ دادن خطاهای نحوی نمی‌شود، بلکه، در واقع هر عملیاتی بر روی متغیری صورت می‌گیرد که مراجعه برای آن یک نام مستعار می‌باشد.مگر اینکه مراجعه به یک ثابت باشد، یک آرگومان ارجاعی بایستی یک *lvalue* (مثلا نام یک متغیر) باشد، و نه یک ثابت یا عبارتی که یک *rvalue* برگشت می‌دهد (مثلا نتیجه ی یک محاسبه). برای آشنایی با تعاریف کلمات *lvalue* و *rvalue* به بخش ۹-۵ مراجعه کنید.

```
1  // Fig. 6.20: fig06_20.cpp
2  // References must be initialized.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     int x = 3;
10    int &y = x; // y refers to (is an alias for) x
11
12    cout << "x = " << x << endl << "y = " << y << endl;
13    y = 7; // actually modifies x
14    cout << "x = " << x << endl << "y = " << y << endl;
15    return 0; // indicates successful termination
16 } // end main
```

```
x = 3
y = 3
x = 7
y = 7
```

شکل ۲۰-۶ | مقداردهی اولیه و استفاده از مراجعه.

```
1  // Fig. 6.21: fig06_21.cpp
2  // References must be initialized.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     int x = 3;
10    int &y; // Error: y must be initialized
```

```
11
12    cout << "x = " << x << endl << "y = " << y << endl;
13    y = 7;
14    cout << "x = " << x << endl << "y = " << y << endl;
15    return 0; // indicates successful termination
16 } // end main
```

Borland C++ command-line compiler error message:

```
Error E2304 C:\cppht5_examples\ch06\Fig06_21\fig06_21.cpp 10:
 Reference variable 'y' must be initialized in function main()
```

Microsoft Visual C++ compiler error message:

```
C:\cppht5_examples\ch06\Fig06_21\fig06_21.cpp(10): error C2530: 'y':
 reference must be initialized
```

GNU C++ compiler error message:

```
fig06_21.cpp(10): error: 'y': declared as a reference but not initialized
```

شکل ٦-٢١ | مراجعه‌ای که مقداردهی اولیه نشده است،سبب تولید خطای نحوی می‌شود.

برگشت دادن یک مراجعه از یک تابع

توابع می‌توانند مراجعه‌ها را برگشت دهند، اما اینکار می‌تواند خطرناک باشد. به هنگام بازگرداندن یک مراجعه به یک متغیر اعلان شده در تابع فراخوانی شده، متغیر باید در درون آن تابع بصورت استاتیک اعلان شود. در غیر اینصورت، مراجعه به یک متغیر اتوماتیک صورت می‌گیرد که با خاتمه یافتن تابع از بین خواهد رفت، چنین متغیری، متغیر "تعریف نشده" گفته می‌شود، و رفتار برنامه غیرقابل پیش بینی می‌شود. مراجعه‌های صورت گرفته به متغیرهای تعریف نشده بعنوان dangling references شناخته می‌شوند.

٦-١٥ آرگومان‌های پیش‌فرض

برای یک برنامه، فراخوانی مکرر یک تابع با مقدار آرگومان یکسان برای یک پارامتر خاص ،یک حالت غیر عادی شمرده نمی‌شود. در چنین مواردی، برنامه‌نویس می‌تواند مشخص کند که چنین پارامتری دارای یک "*آرگومان پیش فرض*" است، یعنی یک مقدار پیش فرض برای ارسال به آن پارامتر. زمانیکه برنامه، آرگومانی را برای پارامتری با آرگومان پیش فرض در فراخوانی تابع در نظر نمی‌گیرد، کامپایلر فراخوانی تابع را بازنویسی کرده و مقدار پیش فرض آن آرگومان را به جای آرگومانی که ارسال نشده است درج می‌کند.

باید آرگومان‌های پیش فرض، سمت راست‌ترین آرگومان در لیست پارامتری یک تابع باشند. در زمان فراخوانی یک تابع با دو یا چندین آرگومان پیش فرض، اگر آرگومان حذف شده سمت راست‌ترین آرگومان در لیست آرگومان نباشد، باید تمامی آرگومان‌های سمت راست آن آرگومان حذف شوند. آرگومان‌های پیش فرض باید در نخستین مکانی که نام تابع آورده شده مشخص شوند (عموما، در نمونه

اولیه تابع). اگر نمونه اولیه تابع به این دلیل که تعریف تابع خود بعنوان نمونه اولیه تابع مطرح است، حذف گردد، بایستی آرگومان‌های پیش فرض در سرآیند تابع مشخص شوند. مقادیر پیش فرض می‌توانند هر عبارتی از جمله ثابت‌ها، متغیرهای سراسری یا فراخوانی‌های تابع باشند. همچنین می‌توان از آرگومان‌های پیش‌فرض در کنار توابع *inline* استفاده کرد.

برنامه شکل ۲۲-۶ به توصیف نحوه استفاده از آرگومان‌های پیش فرض در محاسبه حجم یک جعبه می‌پردازد. نمونه اولیه تابع برای **boxVolume** (خط 8) مشخص می‌کند که هر سه پارامتر مقدار پیش فرض 1 را بدست آورده‌اند. دقت کنید که به منظور افزایش خوانایی، نام متغیرها را در نمونه اولیه تابع قرار داده‌ایم.

```
1  // Fig. 6.22: fig06_22.cpp
2  // Using default arguments.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  // function prototype that specifies default arguments
8  int boxVolume( int length = 1, int width = 1, int height = 1 );
9
10 int main()
11 {
12    // no arguments--use default values for all dimensions
13    cout << "The default box volume is: " << boxVolume();
14
15    // specify length; default width and height
16    cout << "\n\nThe volume of a box with length 10,\n"
17       << "width 1 and height 1 is: " << boxVolume( 10 );
18
19    // specify length and width; default height
20    cout << "\n\nThe volume of a box with length 10,\n"
21       << "width 5 and height 1 is: " << boxVolume( 10, 5 );
22
23    // specify all arguments
24    cout << "\n\nThe volume of a box with length 10,\n"
25       << "width 5 and height 2 is: " << boxVolume( 10, 5, 2 )
26       << endl;
27    return 0; // indicates successful termination
28 } // end main
29
30 // function boxVolume calculates the volume of a box
31 int boxVolume( int length, int width, int height )
32 {
33    return length * width * height;
34 } // end function boxVolume
```

```
The defualt box volume is: 1

The volume of a box with length 10,
withd 1 and height 1 is: 10

The volume of a box with length 10,
withd 5 and height 1 is: 50

The volume of a box with length 10,
withd 5 and height 2 is: 100
```

شکل ۲۲-٦ آرگومان‌های قراردادی در یک تابع.

اولین فراخوانی **boxVolume** (خط 13) آرگومانی را مشخص نکرده است، بنابر این از هر سه مقدار پیش فرض 1 استفاده شده است. در فراخوانی دوم (خط 17) یک آرگومان **length** ارسال شده است، بنابر این از مقادیر پیش فرض 1 برای آرگومان‌های **width** و **height** استفاده شده است. در فراخوانی سوم (خط 21) آرگومان‌هایی برای **length** و **width** ارسال شده است، بنابر این از یک مقدار پیش فرض 1 برای آرگومان **height** استفاده شده است. آخرین فراخوانی (خط 25) آرگومان‌هایی برای **length**، **width** و **height** ارسال کرده است، بنابر این از مقدار پیش فرض استفاده نشده است. دقت کنید که هر آرگومان منتقل شده به تابع صریحاً از چپ به راست به پارامترهای تابع تخصیص داده می‌شود. بنابر این، زمانیکه **boxVolume** یک آرگومان دریافت می‌کند، تابع مقدار آن آرگومان را به پارامتر **length** خود (یعنی سمت چپ‌ترین پارامتر در لیست پارامتری) تخصیص می‌دهد. زمانیکه **boxVolume** دو آرگومان دریافت می‌کند، تابع مقادیر آن آرگومان‌ها را به ترتیب به پارامترهای **length** و **width** خود تخصیص می‌دهد. سرانجام، هنگامی که **boxVolume** سه آرگومان دریافت می‌کند، تابع مقادیر آرگومان‌ها را به ترتیب به پارامترهای **length**، **width** و **height** تخصیص می‌دهد.

برنامه‌نویسی ایده‌ال

استفاده از آرگومان‌های پیش فرض می‌تواند نوشتن فراخوانی‌های تابع را آسان کند. با این وجود، برخی از برنامه‌نویسان احساس می‌کنند که مشخص کردن صریح تمامی آرگومان‌ها واضح‌تر است.

مهندسی نرم‌افزار

اگر مقادیر پیش فرض برای تابعی تغییر پیدا کنند، بایستی کد تمام سرویس‌گیرنده‌ها مجددا کامپایل شود.

خطای برنامه‌نویسی

مبادرت به استفاده از یک آرگومان پیش فرض که سمت راست‌ترین آرگومان نیست، خطای نحوی بدنبال دارد.

۱۶-۶ عملگر تفکیک قلمرو غیرباینری

امکان اعلان متغیرهای محلی و سراسری با نام مشابه وجود دارد. C++ عملگر غیرباینری یا یکانی تفکیک قلمرو (::) را برای دسترسی به یک متغیر سراسری در زمانیکه یک متغیر محلی همنام با آن در قلمرو وجود دارد، تدارک دیده است. عملگر یگانی تفکیک قلمرو نمی‌تواند برای دسترسی به یک متغیر محلی همنام موجود در یک بلوک خارجی بکار گرفته شود. اگر نام متغیر سراسری همنام با یک متغیر محلی در قلمرو نباشد، متغیر سراسری می‌توان مستقیما بدون استفاده از عملگر یگانی تفکیک قلمرو در دسترس قرار گیرد.

برنامه شکل ۲۳-۶ به توصیف عملکرد، عملگر یگانی تفکیک قلمرو با متغیرهای محلی و سراسری همنام (خطوط 7 و 11) پرداخته است. برای تاکید بر اینکه میان نسخه‌های محلی و سراسری متغیر **number**، تمایز وجود دارد، برنامه یک متغیر از نوع **int** و یکی از نوع **double** اعلان کرده است.

```
1  // Fig. 6.23: fig06_23.cpp
2  // Using the unary scope resolution operator.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int number = 7; // global variable named number
8
9  int main()
10 {
11    double number = 10.5; // local variable named number
12
13    // display values of local and global variables
14    cout << "Local double value of number = " << number
15       << "\nGlobal int value of number = " << ::number << endl;
16    return 0; // indicates successful termination
17 } // end main
```

```
Local double value of number = 10.5
Global int value of number = 7
```

شکل ۲۳-٦ | عملگر تفکیک قلمرو غیرباینری.

اجتناب از خطا

از کاربرد متغیرهای همنام برای مقاصد مختلف در یک برنامه، اجتناب کنید. اگرچه می‌توان اینکار را انجام داد، اما این امر می‌تواند خطا ساز شود.

۱۷-٦ سربارگذاری تابع

C++ امکان تعریف توابع همنام، را تا مادامیکه این توابع دارای مجموعه متفاوتی از پارامترها (حداقل در نوع پارامتر یا تعداد پارامترها یا ترتیب نوع پارامترها) باشند، تدارک دیده است. این قابلیت، سربارگذاری تابع نامیده می‌شود. زمانیکه یک تابع سربارگذاری شده فراخوانی می‌شود، کامپایلر C++ با بررسی تعداد، انواع و ترتیب آرگومان‌ها در تابع فراخوانی شده، مبادرت به انتخاب تابع مناسب می‌کند. معمولا از سربارگذاری تابع برای ایجاد چندین تابع همنام که وظایف مشابهی انجام می‌دهند، اما در نوع داده‌ها با هم اختلاف دارند، استفاده می‌شود. برای مثال، بسیاری از توابع در کتابخانه ریاضی برای نوع‌های داده عددی مختلف سربارگذاری شده‌اند.

برنامه‌نویسی ایده‌ال

سربارگذاری توابعی که وظایف مشابهی انجام می‌دهند، می‌تواند سبب افزایش خوانایی و درک برنامه‌ها شود.

سربارگذاری تابع square

برنامه شکل ۲۴-۶ از توابع سربارگذاری شده **square** به منظور محاسبه مربع یک **int** (خطوط 12-8) و مربع یک **double** (خطوط 19-15) استفاده کرده است. خط 23 نسخه **int** از تابع **square** را با ارسال

مقدار لیترال 7 فراخوانی می‌کند. C++ بطور پیش فرض با مقادیر عددی کامل لیترال بصورت نوع int رفتار می‌کند. به همین ترتیب، خط 25 نسخه **double** از تابع **square** را با ارسال مقدار لیترال 7.5 فراخوانی می‌کند، که C++ به طور پیش فرض با آن به شکل یک مقدار **double** رفتار می‌کند. در هر مورد، کامپایلر بر پایه نوع آرگومان مبادرت به انتخاب تابع مناسب برای فراخوانی می‌کند. دو خط آخر از خروجی بر این نکته تاکید می‌کنند که برای هر مورد، تابع صحیح فراخوانی شده است.

```
1  // Fig. 6.24: fig06_24.cpp
2  // Overloaded functions.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  // function square for int values
8  int square( int x )
9  {
10    cout << "square of integer " << x << " is ";
11    return x * x;
12 } // end function square with int argument
13
14 // function square for double values
15 double square( double y )
16 {
17    cout << "square of double " << y << " is ";
18    return y * y;
19 } // end function square with double argument
20
21 int main()
22 {
23    cout << square( 7 ); // calls int version
24    cout << endl;
25    cout << square( 7.5 ); // calls double version
26    cout << endl;
27    return 0; // indicates successful termination
28 } // end main
```

```
square of integer 7 is 49
square of double 7.5 is 56.25
```

شکل ٦-٢٤ | توابع سربارگذاری شده square.

نحوه تفاوت قائل شدن کامپایلر مابین توابع سربارگذاری شده

توابع سربارگذاری شده توسط امضا از یکدیگر متمایز می‌شوند. امضاء ترکیبی از نام تابع و نوع پارامترهای آن است (به ترتیب). کامپایلر شناسه هر تابع را با تعداد و نوع پارامترهای آن رمزگذاری می‌کند تا یک پیوند نوع ایمن (type-safe linkage) بوجود آید. پیوند نوع ایمن ما را مطمئن می‌سازد که تابع سربارگذاری شده صحیح فراخوانی شده و نوع آرگومان‌ها با نوع پارامترها مطابقت دارد.

برنامه شکل ۲۵-۶ با کامپایلر خط فرمان Borland C++ 5.6.4 کامپایل شده است. در این برنامه بجای نشان دادن خروجی حاصل از اجرای برنامه، اسامی تغییر شکل یافته توابع، که توسط زبان اسمبلی در Borland C++ تولید شده‌اند را نشان داده‌ایم. هر نام تغییر شکل یافته با @ شروع و بدنبال آن نام تابع آورده می‌شود. سپس نام تابع با استفاده از **$q** از لیست پارامتر تغییر شکل یافته متمایز می‌شود. در لیست

پارامتری تابع **nothing2** (خط **25**، به چهارمین خط خروجی نگاه کنید)، **c** نشان دهنده یک **char**، **i** نشان دهنده **int**، **rf** نشان دهنده یک **float &** (یعنی یک مراجعه به **float**) و **rd** نشان دهنده یک **double &** (یعنی یک مراجعه به **double**) است. در لیست پارامتری تابع **nothing1**، **i** نشان دهنده یک **int**، **f** نشان دهنده یک **float**، **c** نشان دهنده یک **char** و **ri** نشان دهنده یک **int &** می‌باشد. دو تابع **square** توسط لیست‌های پارامتری خود از یکدیگر متمایز می‌شوند، یکی **d** را برای **double** و دیگری **i** را برای **int** مشخص کرده است. نوع بازگشتی توابع در اسامی تغییر شکل یافته مشخص نشده است. توابع سربارگذاری شده می‌توانند نوع‌های بازگشتی متفاوتی داشته باشند، اما اگر چنین باشد، باید دارای لیست‌های پارامتری متفاوتی هم باشند. از طرف دیگر، نمی‌توانید دو تابع با امضاهای یکسان و نوع‌های بازگشتی متفاوت داشته باشید. دقت کنید که نام تابع تغییر شکل یافته به کامپایلر وابسته است. همچنین دقت کنید که تابع **main** تغییر شکل پیدا نمی‌کند، برای اینکه نمی‌توان آنرا سربارگذاری کرد.

```
1  // Fig. 6.25: fig06_25.cpp
2  // Name mangling.
3
4  // function square for int values
5  int square( int x )
6  {
7     return x * x;
8  } // end function square
9
10 // function square for double values
11 double square( double y )
12 {
13    return y * y;
14 } // end function square
15
16 // function that receives arguments of types
17 // int, float, char and int &
18 void nothing1( int a, float b, char c, int &d )
19 {
20    // empty function body
21 } // end function nothing1
22
23 // function that receives arguments of types
24 // char, int, float & and double &
25 int nothing2( char a, int b, float &c, double &d )
26 {
27    return 0;
28 } // end function nothing2
29
30 int main()
31 {
32    return 0; // indicates successful termination
33 } // end main
```

```
@square$qi
@square$qd
@nothing1$qifcri
@nothing2$qcirfrd
_main
```

۶-۱۸ الگوهای تابع

معمولا توابع سربارگذاری شده برای انجام عملیات‌های مشابهی که منطق متفاوت داشته و بر روی نوع‌های داده متفاوت اعمال می‌گردند، بکار گرفته می‌شوند. اگر منطق و عملیات برنامه برای هر نوع داده با یکدیگر یکسان باشند، می‌توان سربارگذاری را با استفاده از الگوهای تابع به بفرم خلاصه و راحت‌تری انجام داد. برنامه‌نویس یک تعریف منفرد از الگوی تابع را می‌نویسد. با توجه به نوع آرگومان‌های تدارک دیده شده در فراخوانی‌های این تابع، C++ بصورت اتوماتیک مبادرت به تولید الگوی تابع تخصصی مجزا شده به منظور رسیدگی مناسب به هر نوع داده از فراخوانی شده می‌کند. بنابر این، تعریف یک الگوی تابع منفرد، در اصل تعریف کردن کل خانواده توابع سربارگذاری شده است.

برنامه شکل ۲۶-۶ حاوی تعریف یک الگوی تابع (خطوط 18-4) برای تابع **maximum** است که بزرگترین مقدار از میان سه مقدار را مشخص می‌کند. تعریف تمام الگوهای تابع با کلمه کلیدی **template** شروع (خط 4) و بدنبال آن لیست پارامتری الگو در میان جفت کارکتر < > قرار داده می‌شود. هر پارامتر در لیست پارامتری الگو (غالبا *پارامتر نوع رسمی* نامیده می‌شود) همراه با کلمه‌کلیدی **typename** یا **class** آورده می‌شود. پارامترهای نوع رسمی، نقش جانگهدار برای نوع‌های بنیادین یا نوع‌های تعریف شده از سوی کاربر دارند. از این جانگهدارها برای مشخص کردن نوع پارامترهای تابع (خط 5)، نوع برگشتی تابع (خط 5) و اعلان متغیرها در درون بدنه تعریف تابع استفاده می‌شود (خط 7). تعریف یک الگوی تابع همانند سایر توابع است، اما از پارامترهای نوع رسمی بعنوان جانگهدار برای نوع‌های داده واقعی استفاده می‌کند.

```
1  // Fig. 6.26: maximum.h
2  // Definition of function template maximum.
3
4  template < class T >  // or template< typename T >
5  T maximum( T value1, T value2, T value3 )
6  {
7     T maximumValue = value1; // assume value1 is maximum
8
9     // determine whether value2 is greater than maximumValue
10    if ( value2 > maximumValue )
11       maximumValue = value2;
12
13    // determine whether value3 is greater than maximumValue
14    if ( value3 > maximumValue )
15       maximumValue = value3;
16
17    return maximumValue;
18 } // end function template maximum
```

شکل ٢٦-٦ | فایل سرآیند الگوی تابع maximum.

الگوی تابع در برنامه شکل ۲۶-۶ یک پارامتر رسمی بنام **T** (خط 4) و بعنوان یک جانگهدار برای نوع داده‌ای که توسط تابع **maximum** تست خواهد شد، اعلان کرده است. نام یک پارامتر تابع باید در لیست پارامتری الگو منحصر بفرد باشد. زمانیکه کامپایلر تشخیص می‌دهد که تابع **maximum** در کد منبع برنامه احضار شده است، نوع داده ارسالی به **maximum** جانشین **T** در سرتاسر تعریف الگو شده و C++ یک

تابع کامل برای تعیین بزرگترین مقدار از میان سه مقدار از نوع مشخص، ایجاد می‌کند. سپس تابع جدیدا ایجاد شده کامپایل می‌شود. از اینرو، الگوها ابزاری برای تولید کد هستند.

خطای برنامه‌نویسی

قرار ندادن کلمه کلیدی ***class*** *یا* ***typename*** *قبل از هر پارامتر نوع رسمی یک الگوی تابع (مثلا، نوشتن* ***<class S, class T>*** *بجای* ***<class S, T>****) خطای نحوی است.*

برنامه شکل ۲۷-۶ از الگوی تابع **maximum** (خطوط 20، 30 و 40) برای تعیین بزرگترین مقدار در میان سه مقدار صحیح، سه مقدار **double** و سه مقدار **char** استفاده کرده است.

```
1  // Fig. 6.27: fig06_27.cpp
2  // Function template maximum test program.
3  #include <iostream>
4  using std::cout;
5  using std::cin;
6  using std::endl;
7
8  #include "maximum.h" // include definition of function template maximum
9
10 int main()
11 {
12    // demonstrate maximum with int values
13    int int1, int2, int3;
14
15    cout << "Input three integer values: ";
16    cin >> int1 >> int2 >> int3;
17
18    // invoke int version of maximum
19    cout << "The maximum integer value is: "
20       << maximum( int1, int2, int3 );
21
22    // demonstrate maximum with double values
23    double double1, double2, double3;
24
25    cout << "\n\nInput three double values: ";
26    cin >> double1 >> double2 >> double3;
27
28    // invoke double version of maximum
29    cout << "The maximum double value is: "
30       << maximum( double1, double2, double3 );
31
32    // demonstrate maximum with char values
33    char char1, char2, char3;
34
35    cout << "\n\nInput three characters: ";
36    cin >> char1 >> char2 >> char3;
37
38    // invoke char version of maximum
39    cout << "The maximum character value is: "
40       << maximum( char1, char2, char3 ) << endl;
41    return 0; // indicates successful termination
42 } // end main
```

```
Input three integer values: 1 2 3
The maximum integer value is: 3

Input three double values: 3.3 2.2 1.1
The maximum double value is: 3.3

Input three characters: A C B
The maximum character value is: C
```

شکل ۲۷-۶ | عملکرد الگوی تابع maximum.

در برنامه شکل ۲۷-۶، سه تابع بعنوان نتیجه فراخوانی‌های صورت گرفته در خطوط 30 ,20 و 40 ایجاد شده است. در الگوی تابع ایجاد شده برای نوع **int** هر **T** با یک **int** بصورت زیر جایگزین می‌شود:

```
 INT maximum( INT value1, INT value2, INT value3 )
{
   INT maximumValue = value1; // assume value1 is maximum

   // determine whether value2 is greater than maximumValue
   if ( value2 > maximumValue )
      maximumValue = value2;

   // determine whether value3 is greater than maximumValue
   if ( value3 > maximumValue )
      maximumValue = value3;

   return maximumValue;
} // end function template maximum
```

۱۹-۶ بازگشتی

برنامه‌هایی که تا بدین‌جا مطرح شده‌اند عموما از توابعی تشکیل شده بودند که یکدیگر را بصورت سلسله‌مراتبی و منظم فراخوانی می‌کردند. برای حل برخی از مسائل بهتر است که توابع اقدام به فراخوانی خود نمایند. یک تابع بازگشتی می‌تواند بصورت مستقیم یا غیرمستقیم از طریق سایر توابع خود را فراخوانی کند. در این بخش و چند بخش بعدی به بررسی مسائل بازگشتی خواهیم پرداخت.

ابتدا به مفهوم بازگشت، می‌پردازیم و سپس به معرفی چند مثال که در این ارتباط هستند، خواهیم پرداخت. حل مسائل بازگشتی دارای یک سری عناصر مشترک است. توابعی که تا بدین جا مطرح کردیم، برای حل مسائل ساده بودند. در فراخوانی این نوع توابع، تابع به فراخوانی خود پایان داده و کنترل به سادگی به تابع فراخوان باز می‌گردد. یک تابع بازگشتی فراخوانی می‌شود تا مسئله‌ای را حل کند. در واقع تابع فقط از نحوه حل ساده‌ترین حالت یا حالت پایه مطلع است. اگر تابع با حالت پایه فراخوانی شود، تابع نتیجه را برگشت خواهد داد. اگر فراخوانی با مسئله بسیار پیچیده‌ای همراه باشد، تابع مسئله را به دو قسمت مفهومی تقسیم می‌کند: یک قسمت می‌داند که چه کاری می‌خواهد انجام دهد و قسمت بعدی اطلاعی از اینکه چه کاری انجام خواهد داد، ندارد. برای پیاده سازی عمل بازگشتی، قسمت دوم باید با مسئله اصلی شباهت داشته باشد، اما باید بصورت ساده‌تر یا نوع کوچکتر از مسئله اصلی را در برگیرد. بدلیل اینکه این مسئله جدید شبیه مسئله اصلی است، تابع موظف به حل مسئله کوچک و ساده خود است، و این به معنی بازگشتی بودن است و با نام گام بازگشتی هم شناخته می‌شود. گام بازگشتی می‌تواند حاوی کلمه کلیدی

return باشد چرا که نتیجه آن با بخشی از مسئله که تابع از نحوه حل آن مطلع است بکار گرفته خواهد شد. ترکیب این نتایج، سرانجام به فراخوان اصلی ارسال خواهند شد. گام بازگشتی تا زمانیکه تابع اصلی فراخوان، به صورت باز (Open) عمل می‌کند، اجرا می‌شود (اجرا خاتمه نیافته است) ممکن است گام بازگشتی برای کسب نتیجه بارها فراخوانی شود (بصورت بازگشتی) و تابع به زیر مسئله جدیدی در دو قسمت تقسیم شود. در این نوع فراخوان، مسئله در هر بار مکرراً کوچک و کوچکتر می‌شود تا به حالت پایه برسد، از اینرو عمل بازگشتی خاتمه می‌یابد. در این نقطه، تابع حالت پایه را تشخیص داده و نتیجه را برگشت می‌دهد و این فرآیند تا رسیدن به جواب نهایی صورت می‌گیرد. حال به بررسی مثالی می‌پردازیم که دارای مفهوم بازگشتی است، رابطه ریاضی بنام فاکتوریل. فاکتوریل یک عدد غیرمنفی صحیح n که بصورت !n نوشته می‌شود، عبارت است از رابطه

```
n.(n - 1).(n - 2)....1
```

که در آن !1 برابر 1 و !0 برابر 1 تعریف شده است. برای مثال، !5 که بصورت 5×4×3×2×1 نوشته می‌شود حاصلی برابر 120 دارد.

فاکتوریل یک عدد صحیح بزرگتر یا برابر صفر (برای مثال **number**) را می‌توان با روش تکرار (غیربازگشتی) با استفاده از ساختار تکرار **for** پیاده سازی کرد:

```
factorial = 1;
for ( int counter = number; counter >= 1; counter-- )
  factorial *= counter;
```

برای پیاده سازی فاکتوریل به روش بازگشتی، ابتدا باید به رابطه‌ای که در تعریف فاکتوریل وجود دارد توجه کرد:

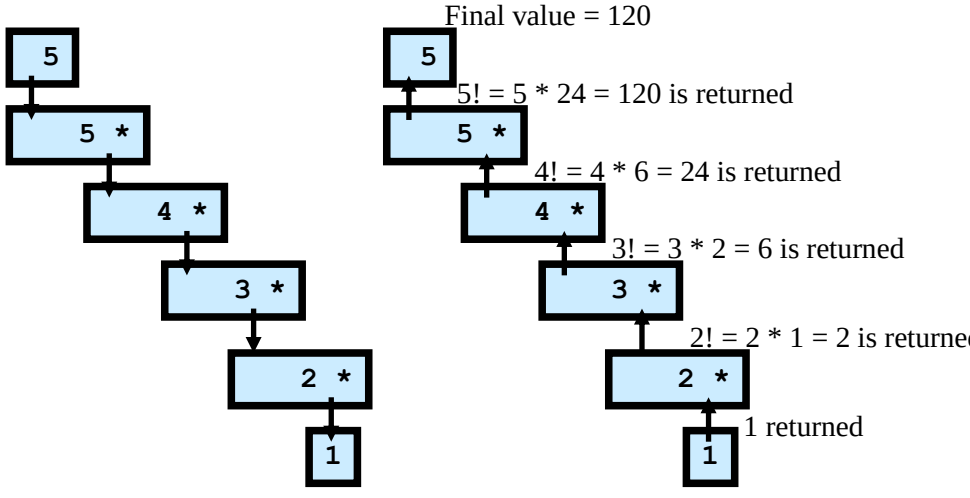
```
n! = n.(n - 1)!
```

برای مثال !5، برابر 5×!4 است و می‌توان آنرا بصورت زیر نشان داد:

```
5! = 5.4.3.2.1
5! = 5.(4.3.2.1)
5! = 4.(4!)
```

ارزیابی !5 در حالت بازگشتی در شکل ۲۸-۶ به نمایش درآمده است. در بخش (a) این شکل، نحوه فراخوانی‌های بازگشتی تا رسیدن به !1 که با 1 ارزیابی می‌شود، دیده می‌شود. در بخش (b) برگشت

مقادیر حاصله از هر فراخوانی بازگشتی به فراخوان خود تا محاسبه آخرین مقدار و برگشت آن، به نمایش درآمده است. برنامه شکل ۲۹-۶ از روش بازگشتی برای محاسبه و چاپ فاکتوریل استفاده می‌کند. تابع بازگشتی **factorial** (خطوط 23-29)، ابتدا تستی را برای تعیین اینکه آیا شرط اتمام حلقه برقرار است یا خیر انجام می‌دهد (مقدار **number** کوچکتر یا برابر **1**). اگر مقدار **number** کوچکتر یا برابر **1** باشد، تابع **factorial**، مقدار **1** را برگشت می‌دهد و انجام فراخوانی‌های بازگشتی بعدی ضرورتی نخواهد داشت. اگر مقدار **number** بزرگتر از **1** باشد، عبارت **number * factorial (number - 1)** اجرا شده و تابع **factorial** بصورت بازگشتی فراخوانی می‌شود تا فاکتوریل **number - 1** محاسبه شود. توجه کنید که **factorial (number - 1)** در واقع حالت ساده شده‌ای از مسئله اصلی که محاسبه **factorial (number)** می‌باشد، است.



(a) Procession of recursive calls. (b) Values returned from each recursive call.

شکل ۲۸-٦ | محاسبه !5 به روش بازگشتی.

```
1  // Fig. 6.29: fig06_29.cpp
2  // Testing the recursive factorial function.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <iomanip>
8  using std::setw;
9
10 unsigned long factorial( unsigned long ); // function prototype
11
12 int main()
```

```
13 {
14    // calculate the factorials of 0 through 10
15    for ( int counter = 0; counter <= 10; counter++ )
16       cout << setw( 2 ) << counter << "! = " << factorial( counter )
17          << endl;
18
19    return 0; // indicates successful termination
20 } // end main
21
22 // recursive definition of function factorial
23 unsigned long factorial( unsigned long number )
24 {
25    if ( number <= 1 ) // test for base case
26       return 1; // base cases: 0! = 1 and 1! = 1
27    else // recursion step
28       return number * factorial( number - 1 );
29 } // end function factorial
```

```
 0! = 1
 1! = 1
 2! = 2
 3! = 6
 4! = 24
 5! = 120
 6! = 720
 7! = 5040
 8! = 40320
 9! = 362880
10! = 3628800
```

شکل ۲۹-٦ | محاسبه فاکتوریل به روش بازگشتی.

تابع **factorial** پارامتری از نوع **unsigned long** دریافت و نتیجه‌ای از نوع **unsigned long** برگشت می‌دهد. این نوع کوتاه شده نماد **unsigned long int** است.بر طبق مستندات C++ استاندارد یک متغیر از نوع **unsigned long int** بایستی حداقل در چهار بایت (**32** بیت) ذخیره شود، از اینرو می‌تواند یک مقدار از 0 تا 4294967295 در خود نگهداری کند. همانطوری که در پنجره خروجی برنامه ۲۹-۶ دیده می‌شود، مقادیر فاکتوریل بسرعت افزایش می‌یابند. بدلیل اینکه نوع داده **unsigned long** فضای زیادی در اختیار دارد، و قادر به نگهداری محاسبه اعداد بزرگتر از !7 است، آنرا انتخاب کرده‌ایم. متأسفانه، مقادیر ایجاد شده توسط تابع **factorial** با سرعتی بزرگ می‌شوند که حتی نوع **unsigned long** هم قادر به نگهداری آنها نمی‌باشد. اینحالت یکی از ضعف‌های بسیار شایع در زبان‌های برنامه‌نویسی است، چرا که نمی‌توان به آسانی در آنها نیازهای منحصر بفرد برنامه‌های مختلف همانند محاسبه مقادیر فاکتوریل اعداد بزرگ را تأمین کرد. همانطوری که شاهد خواهید بود، C++ به عنوان یک زبان گسترش یافته، به برنامه‌نویسان امکان برآوردن احتیاجات منحصر بفرد برنامه‌ها را به کمک نوع داده‌های جدید (بنام کلاس‌ها) فراهم می‌آورد.

خطای برنامه‌نویسی

فراموش کردن حالت پایه یا نوشتن گام بازگشتی که هرگز بحالت پایه نرسد، موجب انجام بازگشت‌های بی‌پایان شده و سرانجام حافظه کاملاً پر خواهد شد.

۲۰-۶ مثال بازگشتی: سری فیبوناچی

سری فیبوناچی بصورت زیر تعریف می‌شود:

0,1,1,2,3,4,8,13,21,...

که با صفر و یک آغاز می‌شود و این خصیصه را دارد که هر عدد بعدی در این سری از مجموع دو عدد قبلی حاصل می‌شود.

این سری بصورت طبیعی رخ داده و در واقع بیان کننده یک فرم حلزونی یا مارپیچی است. نسبت متوالی اعداد فیبوناچی در اطراف مقدار ثابت 1.618 قرار دارد و این عدد به صورت فطری و مداوم تکرار شده و بنام نسبت طلائی مشهور است. تعریف بازگشتی سری فیبوناچی بصورت زیر است:

```
fibonacci(0)= 0
fibonacci(1)= 1
fibonacci(n)= fibonacci(n - 1)+fibonacci(n - 2)
```

دقت کنید که در محاسبه فیبوناچی دو حالت پایه وجود دارد: fibonacci(0) که با مقدار صفر و fibonacci(1) که با مقدار 1 تعریف شده است. در برنامه شکل ۳۰-۶ محاسبه بازگشتی، عدد فیبوناچی i^{th} به کمک تابع **fibonacci** صورت گرفته است. دقت کنید که اعداد فیبوناچی، همانند مقادیر فاکتوریل بسرعت افزایش می‌یابند از اینرو از نوع داده **unsigned long** به عنوان نوع پارامتر و مقدار بازگشتی در تابع **fibonacci** استفاده شده است.

```
1   // Fig. 6.30: fig06_30.cpp
2   // Testing the recursive fibonacci function.
3   #include <iostream>
4   using std::cout;
5   using std::cin;
6   using std::endl;
7
8   unsigned long fibonacci( unsigned long ); // function prototype
9
10  int main()
11  {
12     // calculate the fibonacci values of 0 through 10
13     for ( int counter = 0; counter <= 10; counter++ )
14        cout << "fibonacci( " << counter << " ) = "
15           << fibonacci( counter ) << endl;
16
17     // display higher fibonacci values
18     cout << "fibonacci( 20 ) = " << fibonacci( 20 ) << endl;
19     cout << "fibonacci( 30 ) = " << fibonacci( 30 ) << endl;
20     cout << "fibonacci( 35 ) = " << fibonacci( 35 ) << endl;
21     return 0; // indicates successful termination
22  } // end main
23
24  // recursive method fibonacci
25  unsigned long fibonacci( unsigned long number )
26  {
27     if ( ( number == 0 ) || ( number == 1 ) ) // base cases
28        return number;
29     else // recursion step
30        return fibonacci( number - 1 ) + fibonacci( number - 2 );
```

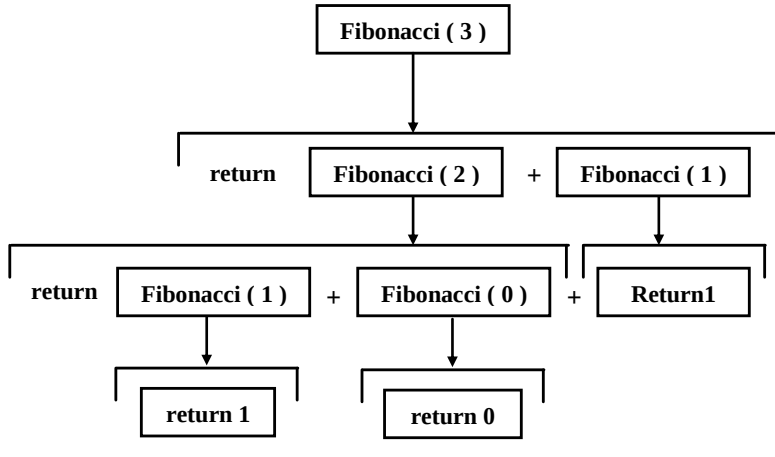
```
31  } // end function fibonacci
```

```
fibonacci(0)= 0
fibonacci(1)= 1
fibonacci(2)= 1
fibonacci(3)= 2
fibonacci(4)= 3
fibonacci(5)= 5
fibonacci(6)= 8
fibonacci(7)= 13
fibonacci(8)= 21
fibonacci(9)= 34
fibonacci(10)= 55
fibonacci(20)= 6765
fibonacci(30)= 832040
fibonacci(35)= 9227465
```

شکل ۳۰-٦ | محاسبه اعداد فیبوناچی به روش بازگشتی.

برنامه با یک عبارت **for** شروع می‌شود که مقادیر فیبوناچی مقادیر 10-0 را محاسبه کرده و بدنبال آن سه فراخوانی برای محاسبه اعداد 30 ,20 و 35 انجام می‌دهد (خطوط 20-18). فراخوانی **fibonacci** (خطوط 19 ,18 ,15 و 20) توسط تابع **main** یک فراخوانی بازگشتی نیست، اما تمام فراخوانی‌های بعدی **fibonacci** (خط30) همگی بازگشتی هستند. هر بار که **fibonacci** اجرا می‌شود، بلافاصله به تست حالت پایه می‌پردازد که به هنگام برابر بودن **number** با 0 یا 1 رخ می‌دهد (خط 27). اگر این شرط برقرار باشد، **number** برگشت داده می‌شود، چرا که fibonacci (0) برابر 0 و fibonacci (1) برابر 1 است. اما در صورتیکه **number** بزرگتر از 1 باشد، گام بازگشتی دو فراخوانی بازگشتی ایجاد می‌کند که هر یک، نوع ساده شده‌ای از مسئله اصلی است.

توجه کنید که برای محاسبه عدد فیبوناچی i^{th}، نیاز به 2^i فراخوانی خواهد بود، پس اگر فیبوناچی 20^{th} محاسبه شود، نیاز به 2^{20} فراخوانی است. شکل ۳۱-۶ نمایشی از نحوه ارزیابی **fibonacci(3)** است.



شکل ۳۱-٦ | فراخوانی‌های بازگشتی تابع fibonacci.

۲۱-٦ بازگشتی یا تکرار

در دو بخش قبلی دو تابع را دیدیم که می‌توانستند بسادگی وظایف خود را با استفاده از روش بازگشتی یا تکرار انجام دهند. در این بخش، این دو روش را باهم مقایسه کرده و در مورد اینکه چرا در برخی از مواقع یکی از این روش‌ها به روش دیگری ترجیح داده می‌شود، بحث خواهیم کرد.

هر دو روش تکرار و بازگشتی، بر مبنی ساختارهای کنترل هستند، تکرار از ساختارهای تکرار شونده همانند **for** یا **while** استفاده می‌کند، در حالیکه روش بازگشتی از ساختارهای انتخاب همانند **if**، **if…else** یا **switch** سود می‌برد. اگر چه هر دو روش مستلزم تکرار هستند، اما روش تکرار بصورت صریح از ساختارهای تکرار استفاده می‌کند در حالیکه روش بازگشتی از طریق فراخوانی‌های مکرر، عمل تکرار را انجام می‌دهد. تکرار و هم بازگشتی هر کدام مستلزم انجام یک تست خاتمه دهنده هستند. در تکرار هنگامی که شرط تکرار حلقه برقرار نباشد، تکرار خاتمه می‌یابد و در بازگشتی این اتمام به هنگام تشخیص حالت اصلی (پایه) صورت می‌گیرد.

در روش تکرار با استفاده از شمارنده-کنترل تکرار به طرف خاتمه حلقه حرکت می‌کنیم و در بازگشتی بصورت تدریجی به خاتمه نزدیک می‌شویم. در تکرار تغییرات شمارنده در نظر گرفته می‌شود تا شمارنده با یک مقدار از قبل تعیین شده، شرط تکرار حلقه را نقض کند و در بازگشتی با حفظ مسئله ساده شده از مسئله اصلی و ادامه آن تا رسیدن به حالت پایه. هم تکرار و هم بازگشتی می‌توانند بصورت نامحدود ادامه داشته باشند یک حلقه بی‌نهایت می‌تواند اگر شرط حلقه هیچ‌گاه برقرار نشود، تا بی‌نهایت تکرار شود، و در بازگشتی اگر گام بازگشتی نتواند در هر بار مسئله را ساده‌تر نماید و بحالت پایه برساند، فراخوانی بی‌نهایت بار اتفاق می‌افتد.

برای بیان تفاوت‌های موجود مابین روش تکرار و بازگشتی، اجازه دهید تا به بررسی راه‌حل تکرار برای مسئله فاکتوریل بپردازیم (شکل ۳۲-۶). دقت کنید که از یک عبارت تکرار (خطوط 29-28 از شکل ۳۲-۶) بجای عبارت انتخاب در راه‌حل بازگشتی مسئله استفاده کرده‌ایم (خطوط 27-24 از شکل ۲۹-۶). توجه کنید که در هر دو راه حل از یک تست خاتمه استفاده شده است. در روش بازگشتی، خط 24 تستی برای حالت پایه انجام می‌دهد. در روش تکرار، خط 28 تستی بر روی شرط حلقه انجام می‌دهد، اگر شرط برقرار نباشد،

حلقه خاتمه می‌پذیرد. در پایان دقت کنید که بجای تولید یک نسخه ساده از مسئله اصلی، راه حل تکرار از یک شمارنده تغییر پذیر استفاده کرده تا اینکه شرط تکرار حلقه برقرار نشود.

```
1  // Fig. 6.32: fig06_32.cpp
2  // Testing the iterative factorial method.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <iomanip>
8  using std::setw;
9
10 unsigned long factorial( unsigned long ); // function prototype
11
12 int main()
13 {
14    // calculate the factorials of 0 through 10
15    for ( int counter = 0; counter <= 10; counter++ )
16       cout << setw( 2 ) << counter << "! = " << factorial( counter )
17          << endl;
18
19    return 0;
20 } // end main
21
22 // iterative method factorial
23 unsigned long factorial( unsigned long number )
24 {
25    unsigned long result = 1;
26
27    // iterative declaration of method factorial
28    for ( unsigned long i = number; i >= 1; i-- )
29       result *= i;
30
31    return result;
32 } // end function factorial
```

```
 0! = 1
 1! = 1
 2! = 2
 3! = 6
 4! = 24
 5! = 120
 6! = 720
 7! = 5040
 8! = 40320
 9! = 362880
10! = 3628800
```

شکل ۳۲-٦ | راه حل تکرار برای تابع فاکتوریل.

بازگشتی دارای معایب زیادی است، مکانیزم فراخوانی‌های متعدد و در نتیجه فراخوانی‌های بسیار زیاد تابع یکی از معایب آن است و این امر می‌تواند برای زمان پردازنده و حافظه گران تمام شود. هر فراخوانی بازگشتی موجب می‌شود تا یکی کپی از تابع (همراه با داده‌ها) تهیه شود و خود این عمل در مصرف حافظه بسیار موثر است. اما در تکرار چنین اتفاقاتی رخ نمی‌دهد. پس چرا بازگشتی را انتخاب می کنیم؟

مهندسی نرم‌افزار

هر مسئله‌ای که با استفاده از روش بازگشتی حل می‌شود، می‌تواند بصورت غیربازگشتی هم حل شود.

بازگشتی یک انتخاب و نزدیک شدن به مسئله بصورت عادی است در حالیکه در تکرار به مسئله بصورت فطری یا ذاتی نزدیک شده و فهم و خطایابی برنامه بسیار آسانتر است. همچنین زمانی از روش بازگشتی استفاده می‌شود که روش تکرار برای حل مسئله مناسب نیست.

کارایی

سعی کنید از بکار بردن روش بازگشتی در حل مسائل خودداری کنید چرا که فراخوانی‌های متعدد باعث افزایش زمان و مصرف حافظه می‌شوند.

مثال ها و تمرین های بازگشت	در کتاب
تابع فاکتوریل تابع فیبوناچی مجموع دو عدد صحیح به توان رساندن دو عدد صحیح با یک عدد صحیح برج های هانوی تصور بازگشت بزرگترین مقسوم علیه مشترک "این برنامه چه کاری انجام می دهد؟"	فصل ٦ بخش۱۹-٦، شکل ۲۹-٦ بخش۱-٦۹، شکل ۳۰-٦ خودآزمایی ۷-٦ تمرین ٤۰-٦ تمرین ٤۲-٦ تمرین ٤٤-٦ تمرین٤٥-٦ تمرین٥۰-٦، تمرین٥۱-٦
"این برنامه چه کاری انجام می دهد؟" "این برنامه چه کاری انجام می دهد؟" مرتب سازی انتخابی تعیین متقارن بودن یک رشته جستجوی خطی جسجوی دودویی هشت وزیر چاپ یک آرایه چاپ یک رشته به ترتیب عکس کوچکترین مقدار موجوددر یک آرایه	فصل ۷ تمرین ۱۸-۷ تمرین۲۱-۷ تمرین۳۱-۷ تمرین۳۲-۷ تمرین۳۳-۷ تمرین۳٤-۷ تمرین۳٥-۷ تمرین۳٦-۷ تمرین۳۷-۷ تمرین۳۸-۷
مرتب سازی سریع پیمایش maze تولید maze به صورت تصادفی Maze در سایزهای مختلف	فصل ۸ تمرین۲٤-۸ تمرین۲٥-۸ تمرین۲٦-۸ تمرین۲۷-۸
مرتب سازی ادغامی جستجوی خطی	فصل ۲۰ بخش ۳-۳-۲۰، شکل ٥-۲۰ تا ۷-۲۰

جستجوی دودویی **مرتب سازی سریع**	**تمرین۸-۲۰** **تمرین۹-۲۰** **تمرین۱۰-۲۰**
افزودن عضو به درخت دودویی **پیمایش پیش ترتیب یک درخت دودویی** **پیمایش میان ترتیب یک درخت دودویی** **پیمایش پس ترتیب یک درخت دودویی** **چاپ یک لیست پیوندی به ترتیب بر عکس** **جستجوی یک لیست پیوندی** **حذف درخت دودویی** **چاپ درخت**	**فصل ۲۱** **بخش ۷-۲۱-شکل ۲۰-۲۱ تا ۲۲-۲۱** **بخش ۷-۲۱-شکل ۲۰-۲۱ تا ۲۲-۲۱** **بخش ۷-۲۱-شکل ۲۰-۲۱ تا ۲۲-۲۱** **بخش ۷-۲۱ شکل ۲۰-۲۱ تا ۲۲-۲۱** **تمرین۲۰-۲۱** **تمرین۲۱-۲۱** **تمرین۲۲-۲۱** **تمرین۲۵-۲۱**

شکل ۳۳-٦ | مثال‌های بازگشتی و تمرینات مرتبط با آن.

۲۲-۶ مبحث آموزشی مهندسی نرم‌افزار: شناسایی عملیات‌های کلاس در سیستم ATM

در بخش‌های "مبحث آموزشی مهندسی نرم‌افزار" در انتهای فصل‌های سوم، چهارم و پنجم، قدم‌های اولیه در طراحی شی‌گرا سیستم ATM را برداشتیم. در این بخش، به تعیین برخی از عملیات‌های کلاس (یا رفتارها) مورد نیاز در پیاده‌سازی سیستم ATM می‌پردازیم.

شناسایی عملیات‌ها

یک عملیات، سرویسی است که شی‌های یک کلاس به سرویس‌گیرنده‌های کلاس عرضه می‌کنند. به عملیات برخی از شی‌ها در دنیای واقعی توجه کنید. عملیات یک رادیو شامل تنظیم ایستگاه و صدای آن است (معمولا توسط شخصی که کنترل‌های رادیو را تنظیم می‌کند، انجام می‌شوند). عملیات یک اتومبیل شامل شتاب‌گیری (توسط راننده و با فشار دادن پدال گاز)، کاهش شتاب (توسط راننده و با فشار دادن پدال ترمز یا رها کردن پدال گاز)، چرخش و تعویض دنده‌ها است. شی‌ها نرم‌افزاری هم عملیات‌های گوناگونی انجام می‌دهند، برای مثال، یک نرم‌افزار گرافیکی می‌تواند عملیات‌های برای ترسیم دایره، خط، مربع و کارهای دیگر انجام دهد. نرم‌افزار صفحه‌گسترده می‌تواند عملیاتی مانند چاپ صفحه‌گسترده، جمع عناصر در سطر و ستون و چاپ نمودار بصورت میله‌ای یا دایره‌ای انجام دهد.

می‌توانیم با بررسی فعل‌های کلیدی و عبارات فعلی در مستند نیازها، تعدادی از عملیات هر کلاس را استنتاج کنیم. سپس هر یک از آنها را به کلاس‌های خاصی در سیستم خود مرتبط می‌کنیم (شکل ۳۴-۶). عبارات یا جمله‌های فعلی به نمایش در آمده در جدول شکل ۳۴-۶ در تعیین عملیات‌های هر کلاس به ما کمک می‌کنند.

مدلسازی عملیات‌ها

برای شناسایی عملیات‌ها، به بررسی عبارات فعلی لیست شده برای هر کلاس در جدول شکل ۳۴-۶ می‌پردازیم. عبارت "اجرای تراکنش مالی" مرتبط با کلاس **ATM** بطور ضمنی نشان می‌دهد که کلاس **ATM** به تراکنش دستور اجرا شدن صادر می‌کند. بنابر این، کلاس‌های **BalanceInquiry**، **Withdrawal** و **Deposit** هر کدام به یک عملیات برای تدارک دیدن این سرویس برای ATM نیاز دارند. ما این عملیات (که آن را **execute** نام داده‌ایم) را در قسمت سوم از کلاس‌های سه‌گانه تراکنشی در دیاگرام به روز شده شکل ۳۵-۶ قرار داده‌ایم. در مدت زمان یک جلسه ATM، شی **ATM** عملیات **execute** را برای هر شی تراکنشی فراخوانی می‌کند تا به اجرا در آیند.

زبان UML عملیات‌ها را (که بعنوان توابع عضو در C++ پیاده‌سازی می‌شوند) با لیست کردن نام عملیات، و بدنبال آن لیست پارامتری جدا شده با کاما در درون پرانتزها، یک کولن و نوع بازگشتی عرضه می‌کند:

نوع بازگشتی : (پارامتر n ،... ،پارامتر ۲ ،پارامتر ۱) نام عملیات

هر پارامتر در این لیست متشکل از یک نام پارامتر، بدنبال آن یک کولن و نوع پارامتر است:

نوع پارامتر : نام پارامتر

در این بخش مبادرت به لیست کردن پارامترهای عملیات خود نکرده‌ایم، بزودی به شناسایی و مدل کردن پارامترهای برخی از عملیات‌ها اقدام خواهیم کرد. برای برخی از عملیات‌ها، هنوز اطلاعی از نوع برگشتی آنها نداریم، از اینرو در دیاگرام خبری از آنها نیست. در این مرحله از طراحی عدم حضور آنها کاملا عادی است. همانطوری که فرآیند پیاده‌سازی به جلو می‌رود، نوع‌های بازگشتی باقیمانده را به دیاگرام اضافه خواهیم کرد.

کلاس	افعال و عبارات فعلی
`ATM`	اجرای تراکنش مالی
`BalanceInquiry`	{در مستند نیازها وجود ندارد}
`WithDrawal`	{در مستند نیازها وجود ندارد}
`Deposit`	{در مستند نیازها وجود ندارد}
`BankDatabase`	احراز هویت کاربر، بازیابی موجودی حساب، میزان سپرده‌گذاری در حساب، بدهکار کردن حساب به میزان برداشت پول
`Account`	بازیابی موجودی حساب، میزان سپرده‌گذاری در حساب، بدهکار کردن حساب به میزان برداشت پول
`Screen`	نمایش پیغام به کاربر
`Keypad`	دریافت ورودی عددی از کاربر
`CashDispenser`	پرداخت پول، تعیین اینکه به میزان کافی پول نقد برای پاسخ به تقاضا وجود دارد یا خیر

DepositSlot	دریافت پاکت سپرده

شکل ٦-٣٤ | افعال و عبارات فعلی برای هر کلاس در سیستم ATM.

شکل ٦-٣٥ | کلاس‌ها در سیستم ATM به همراه صفات و عملیات‌ها.

عملیات کلاس‌های BankDatabase و Account

در جدول شکل ۳۴-۶ جمله "احراز هویت کاربر" در کنار کلاس **BankDatabase** قرار داده شده است، پایگاه داده شی است حاوی اطلاعات حساب مورد نیاز برای تعیین اینکه آیا شماره حساب و **PIN** وارد شده از طرف کاربر مطابق با شماره حساب و **PIN** نگه‌داری شده در بانک است یا خیر. بنابر این، کلاس **BankDatabase** به عملیاتی نیاز دارد که سرویس احراز هویت را برای **ATM** تدارک ببیند. عملیات **authenticateUser** را در قسمت سوم از کلاس **BankDatabase** جای داده‌ایم (شکل ۳۵-۶). با این وجود، یک شی از کلاس **Account**، و نه کلاس **BankDatabase**، مبادرت به ذخیره شماره حساب و **PIN** می‌کند که بایستی برای احراز هویت کاربر در دسترس باشد، از اینرو کلاس **Accoount** باید سرویسی برای اعتبارسنجی **PIN** وارد شده از سوی کاربر با **PIN** ذخیره شده در یک شی **Account** تدارک دیده باشد. بنابر این، عملیات **validatePIN** را برای کلاس **Account** در نظر گرفته‌ایم. توجه کنید که نوع برگشتی **Boolean** را برای عملیات‌های **authenticateUser** و **validatePIN** انتخاب کرده‌ایم. هر عملیاتی یک مقدار برگشت می‌دهد و این مقدار دلالت بر این دارد که آیا عملیات در انجام وظیفه خود موفق بوده (یعنی برگشت مقدار **true**)یا خیر (یعنی برگشت مقدار **false**).

در جدول شکل ۳۴-۶ چندین عبارت فعلی برای کلاس **BankDatabase** لیست شده است: "بازیابی موجودی حساب"، "میزان سپرده‌گذاری در حساب"، "بدهکار کردن حساب به میزان برداشت پول". همانند "احراز هویت کاربر" این عبارات فعلی اشاره به سرویس‌های دارند که بایستی پایگاه داده برای **ATM** تدارک ببیند، چرا که پایگاه داده تمام اطلاعات حساب را که اعتبارسنجی کاربر و تراکنش‌های **ATM** نقش دارند در خود ذخیره کرده است. با این وجود، در واقع شی‌های از کلاس **Account** عملیات‌های که این عبارات فعلی به آنها اشاره دارند، را انجام می‌دهند. از اینرو، یک عملیات به هر دو کلاس **BankDatabase** و **Account** تخصیص داده‌ایم تا متناظر با هر یک از این عبارات باشند. از بخش ۱۱-۳ به یاد دارید که چون، یک حساب بانکی حاوی اطلاعات حساس است، به **ATM** اجازه دسترسی مستقیم به حساب را ندادیم. پایگاه داده بصورت یک میانجی یا واسط مابین **ATM** و داده‌های حساب عمل می‌کند، بنابر این جلوی دسترسی غیرمجاز گرفته می‌شود. همانطوری که در بخش ۱۲-۷ مشاهده خواهید کرد، کلاس **ATM** عملیات‌های کلاس **BankDatabase** را فراخوانی می‌کند، که هر یک از آنها در ادامه عملیاتی همنام را در کلاس **Account** فراخونی می‌کنند.

جمله "بازیابی موجودی حساب" نشان می‌دهد که کلاس‌های **BankDatabase** و **Account** به عملیات **getBalance** نیاز دارند. با این همه، بخاطر دارید که دو صفت برای کلاس **Account** به منظور نمایش موجودی ایجاد کردیم، **availableBalance** و **totalBalance**. پرس‌وجوی یک موجودی مستلزم دسترسی به هر دو صفت موجودی است، از اینرو است که می‌تواند آنها را به کاربر نشان دهد، اما برداشت پول فقط نیاز به بررسی مقدار **availableBalance** دارد. برای اجازه دادن به شی‌ها در سیستم برای بدست آوردن هر صفت موجودی بصورت مجزا از هم، مبادرت به افزودن عملیات‌های **getAvailableBalance** و **getTotalBalance** به بخش سوم از کلاس‌های **BankDatabase** و **Account** کرده‌ایم (شکل ۳۵-۶). نوع برگشتی را از نوع **Double** برای هر یک از این عملیات‌ها در نظر گرفته‌ایم، چرا که صفات موجودی که آنها بازیابی می‌کنند از نوع **Double** هستند.

جمله " میزان سپرده‌گذاری در حساب" و "بدهکار کردن حساب به میزان برداشت پول" نشان می‌دهند که کلاس‌های **BankDatabase** و **Account** باید عملیاتی برای به روز کردن یک حساب در جریان سپرده‌گذاری و برداشت پول انجام دهند. از اینرو، مبادرت به تخصیص عملیات‌های **credit** و **debit** به کلاس‌های **BankDatabase** و **Account** کرده‌ایم. بخاطر دارید که دادن اعتبار به حساب فقط مقداری را به صفت **totalBalance** اضافه می‌کند. از طرف دیگر، بدهکار کردن یک حساب (در نتیجه برداشت پول) از میزان هر دو صفت موجودی کم می‌کند. این جزئیات پیاده‌سازی را در درون کلاس **Account** پنهان کرده‌ایم. اینحالت مثال خوبی از کپسوله‌سازی و پنهان‌سازی اطلاعات است.

عملیات کلاس Screen

کلاس **Screen** در زمان‌های مختلف در یک جلسه **ATM** "پیغام‌های را به کاربر نشان می‌دهد". تمامی خروجی‌های بصری از طریق صفحه‌نمایش **ATM** رخ می‌دهند. در مستند نیازها انواع مختلفی از پیغام‌ها آورده شده است (همانند، پیغام خوش‌آمدگویی، پیغام خطا، پیغام تشکر) که صفحه‌نمایش برای کاربر بنمایش در می‌آورد. همچنین مستند نیازها نشان می‌دهد که صفحه‌نمایش اعلان‌ها و منوهای را به کاربر نشان می‌دهد. با این همه، اعلان در واقع یک پیغام توضیحی است که به کاربر آنچه را که باید انجام دهد، دیکته می‌کند و منو اصولا نوعی اعلان از چندین پیغام (گزینه‌های منو) است. بنابر این، بجای تخصیص اختصاصی کلاس **Screen** به هر عملیات برای نمایش هر نوع پیغام، اعلان و منو، فقط یک عملیات ایجاد می‌کنیم که می‌تواند هر پیغام مشخص شده توسط پارامتر را به نمایش در آورد. این عملیات را که **displayMessage** نام دارد، در قسمت سوم از کلاس **Screen** دیاگرام کلاس خود جای داده‌ایم (شکل ۳۵-۶). توجه کنید که فعلا نگران پارامترهای این عملیات نیستیم، در انتهای این بخش مبادرت به مدل کردن پارامتر خواهیم کرد.

عملیات کلاس Keypad

از جمله "دریافت ورودی عددی از کاربر" در جدول شکل ۳۴-۶ چنین برداشت می‌کنیم که بایستی کلاس **Keypad** عملیات **getInput** را انجام دهد. چون صفحه کلید **ATM**، برخلاف صفحه کلید کامپیوتر، فقط حاوی اعداد **0-9** است، تعیین کرده‌ایم که این عملیات یک مقدار صحیح برگشت دهد. از مستند نیازها بخاطر دارید که در شرایط مختلف، امکان دارد کاربر ارقام متفاوتی وارد سازد (مثلا، شماره حساب، **PIN**، شماره گزینه منو، میزان سپرده‌گذاری). کلاس **Keypad** بسادگی مقدار عددی را برای سرویس‌گیرنده کلاس بدست می‌آورد، این کلاس مسئول تست مقدار وارد شده تحت ضوابط خاص نیست. هر کلاسی که از این عملیات استفاده می‌کند باید به اعتبارسنجی مقدار وارد شده از سوی کاربر پرداخته و در صورت اشتباه بودن، پیغام مناسب خطا را از طریق کلاس **Screen** بنمایش در آورد.

عملیات کلاس‌های CashDispenser و DepositSlot

در جدول شکل ۳۴-۶ عبارت "پرداخت پول" برای کلاس **CashDispenser** لیست شده است. بنابر این، عملیات **dispenseCash** را ایجاد و آنرا تحت کلاس **CashDispenser** در شکل ۳۵-۶ لیست کرده‌ایم. همچنین کلاس **CashDispenser** حاوی عبارت "تعیین اینکه به میزان کافی پول نقد برای پاسخ به تقاضا وجود دارد یا خیر" است. از اینرو، **isSufficientCashAvailable** را بعنوان عملیاتی که مقداری از نوع **Boolean** در کلاس **CashDispenser** برگشت می‌دهد، در نظر گرفته‌ایم. همچنین در جدول شکل ۳۴-۶ عبارت "دریافت پاکت سپرده‌گذاری" برای کلاس **DepositSlot** لیست شده است. بایستی شکاف سپرده‌گذاری تعیین کند که آیا پاکتی دریافت کرده است یا خیر، از اینرو عملیات **isEnvelopeReceived** را در نظر گرفته‌ایم، که مقداری از نوع **Boolean** در بخش سوم کلاس **DepositSlot** برگشت می‌دهد.

عملیات کلاس ATM

در این لحظه عملیاتی برای کلاس **ATM** لیست نشده است. هنوز از سرویس‌های که کلاس **ATM** برای سایر کلاس‌ها در سیستم تدارک دیده است، اطلاعی نداریم. زمانیکه، سیستم را با کد C++ پیاده‌سازی می‌کنیم، عملیات‌های این کلاس به همراه چندین عملیات دیگر از سایر کلاس‌ها به سیستم خواهند پیوست.

شناسایی و مدل کردن پارامترهای عملیاتی

تا بدین مرحله، توجهی به پارامترها در عملیات‌های خود نداشتیم. حال اجازه دهید از نزدیک نگاهی به پارامترهای عملیاتی بیندازیم. یک پارامتر عملیاتی را با بررسی داده مورد نیاز عملیات برای انجام وظیفه در نظر گرفته شده، شناسایی می‌کنیم.

به عملیات **authenticateUser** در کلاس **BankDatabase** توجه کنید. برای احراز هویت یک کاربر، بایستی این عملیات از شماره حساب و **PIN** تدارک دیده شده از سوی کاربر مطلع باشد. از اینرو، مشخص کرده‌ایم که عملیات **authenticateUser** پارامترهای صحیح **userAccountNumber** و **userPIN** را دریافت کند، که عملیات باید به مقایسه شماره حساب و **PIN** از شی **Account** در پایگاه داده بپردازد. در ابتدای نام این پارامترها پیشوند "user" را قرار داده‌ایم تا از اشتباه شدن مابین اسامی پارامتر عملیات و اسامی صفت که متعلق به کلاس **Account** هستند، اجتناب شود. در شکل ۳۶-۶ این پارامترها را در دیاگرام کلاس لیست کرده‌ایم، که فقط مدل کننده کلاس **BankDatabase** است.

شکل ٣٦-٦ | کلاس BankDatabase با پارامترهای عملیاتی.

بخاطر دارید که UML هر پارامتر را در یک لیست پارامتری مجزا شده با کاما که متعلق به یک عملیات است، با لیست کردن نام پارامتر، بدنبال آن یک کولن و نوع پارامتر مدل‌سازی می‌کند. از اینرو، در شکل ۳۶-۶ مشخص است که عملیات **authenticateUser** دو پارامتر دریافت می‌کند: **userAccountNumber** و **userPIN** که هر دو از نوع **Integer** هستند. زمانیکه سیستم را در C++ پیاده‌سازی کردیم، این پارامترها با مقادیر **int** عرضه خواهند شد.

عملیات‌های **getAvailableBalance, getTotalBalance, credit** و **debit** از کلاس **BankDataBase** نیز نیازمند پارامتر **userAccountNumber** برای شناسایی حسابی هستند که باید پایگاه داده بر روی آن عملیات را اجرا کند، از اینرو این پارامترها را در دیاگرام کلاس شکل ۳۶-۶ وارد کرده‌ایم. علاوه بر این، عملیات‌های **credit** و **debit** هر یک نیازمند پارامتر **amount** از نوع **Double** هستند تا تعیین کننده پول به اعتبار گذاشته شده یا بدهی باشد.

دیاگرام کلاس در شکل ۳۷-۶ مدل کننده پارامترهای عملیاتی کلاس **Account** است. عملیات **validatePIN** فقط نیازمند یک پارامتر بنام **userPIN** است، که حاوی **PIN** وارد شده از سوی کاربر برای مقایسه شدن با **PIN** متناظر در حساب است. همانند همکارهای خود در کلاس **BankDatabase**، عملیات‌های **credit** و **debit** در کلاس **Account** هر یک نیازمند پارامتر **amount** از نوع **Double** هستند.

شکل ٣٧-٦ | کلاس Account با پارامترهای عملیاتی.

عملیات‌های **getAvailableBalance** و **getTotalBalance** در کلاس **Account** نیازی به داده اضافی برای انجام وظایف خود ندارند.

شکل ۳۸-۶ کلاس **Screen** را با پارامتر مشخص شده برای عملیات **displayMessage** مدل کرده است. این عملیات فقط مستلزم یک پارامتر بنام **message** از نوع **String** است که دلالت بر پیغام بنمایش در آمده دارد.

دیاگرام کلاس در شکل ۳۹-۶ مشخص می‌کند که عملیات **dispenseCash** از کلاس **CashDispenser** یک پارامتر بنام **amount** از نوع **Double** دریافت می‌کند که نشاندهنده پول نقد است. همچنین عملیات **isSufficientCashAvailable** پارامتر **amount** را که از نوع **Double** است دریافت می‌کند، که نشاندهنده مقدار پول در درخواست است.

توجه کنید که بحثی در ارتباط با پارامترهای عملیاتی **execute** کلاس‌های **BalanceInquiry, Withdrawal** و **Deposit**، عملیات **getInput** از کلاس **Keypad** و عملیات **isEnvelopeReceived** از کلاس **DepositSlot** به میان نیاوردیم. تا بدین مرحله از فرآیند طراحی، نمی‌توانیم تعیین کنیم که آیا این عملیات‌ها نیازمند داده‌های اضافی برای انجام وظایف خود هستند یا خیر، بنابر این لیست پارامتری آنها را خالی نگه داشتیم. همانطوری که به پیش می‌رویم، در مورد افزودن پارامترها به عملیات‌ها تصمیم می‌گیریم.

شکل ۳۸-٦ | کلاس Screen با پارامترهای عملیاتی.

شکل ۳۹-٦ | کلاس CashDispenser با پارامترهای عملیاتی.

تمرینات خودآزمایی مبحث مهندسی نرم‌افزار

٦-١١ کدام یک از موارد زیر یک رفتار نیست؟.

a) خواندن داده از یک فایل

b) چاپ خروجی

c) خروجی متنی

d) بدست آوردن ورودی از کاربر

٦-٢ اگر بخواهید عملیاتی به سیستم ATM اضافه کنید که صفت amount از کلاس Withdrawal را برگشت دهد، چگونه و در کجای دیاگرام شکل ۳۵-۶ اینکار را انجام می‌دهید.

٦-٣ مفهوم عملیات لیست شده در زیر را که ممکن است در دیاگرام کلاسی بکار رفته باشد، چیست؟

```
add( x : Integer, y : Integer ) : Integer
```

پاسخ خودآزمایی مبحث آموزشی مهندسی نرم‌افزار

٦-١ c

٦-٢ می‌توان این عملیات را در قسمت سوم کلاس **Withdrawal** جای داد:

```
getAmount() : Double
```

۶-۳ نام این عملیات add بوده و پارامترهای صحیح x و y را دریافت کرده و یک مقدار صحیح برگشت می‌دهد.

خودآزمایی

۶-۱ جاهای خالی را در عبارات زیر با کلمات مناسب پر کنید.

a) کامپونت‌های برنامه در C++، ..... و ..... نامیده می‌شوند.

b) یک تابع با ..... آن فعال می‌شود.

c) متغیری که فقط در درون تابع اعلان شده شناخته شود، متغیری از نوع ..... نامیده می‌شود.

d) عبارت ..... در تابع فراخوانی شده، موجب ارسال مقداری به تابع فراخواننده می‌شود.

e) تابعی که با کلمه کلیدی ..... تعریف شده باشد، مقدار باز نمی‌گرداند.

f) ..... یک شناسه بخشی از برنامه است که در آن بخش شناسه قابل استفاده می‌باشد.

g) به سه روش می‌توان کنترل را از یک تابع فراخوانی شده به فراخوان بازگرداند، این سه روش عبارتنداز .....، ..... و ..... .

h) یک ..... به کامپایلر اجازه بررسی تعداد، نوع‌ها و ترتیب آرگومان‌های ارسالی به تابع را می‌دهد.

i) تابع ..... اعداد تصادفی ایجاد می‌کند.

j) تابع ..... برای تنظیم عدد تصادفی و تغذیه آن بکار گرفته می‌شود.

k) تصریح کننده‌های کلاس ذخیره‌سازی عبارتند از mutable، .....، .....، ..... و .....

l) متغیرهای اعلان شده در یک بلوک یا لیست پارامترهای تابع دارای کلاس ذخیره‌سازی ..... هستند.

m) تصریح کننده کلاس ذخیره‌سازی ..... به کامپایلر توصیه می‌کند که متغیر را در ثبات کامپیوتر ذخیره کند.

n) متغیر اعلان شده در خارج از هر بلوک یا تابع، متغیر ..... نامیده می‌شود.

o) باید متغیر محلی در تابعی را که مقدار خود را مابین فراخوانی تابع حفظ می‌کند، بصورت ..... در کلاس ذخیره‌سازی اعلان شود.

p) شش قلمرو ممکنه برای یک شناسه عبارتند از .....، .....، .....، .....، ..... و ..... .

q) تابعی که خود را بصورت مستقیم یا غیرمستقیم فراخوانی می‌کند یک تابع ..... است.

r) یک تابع بازگشتی عموماً متشکل از دو کامپونت است: بخشی که منظور از آن اتمام رفتار بازگشتی با تست حالت ..... است و بخشی که مسئله را به فرم بازگشتی مطرح و فراخوانی می‌کند.

s) در C++، امکان داشتن توابع مضاعف با یک نام وجود دارد که در آنها نوع یا تعداد آرگومان متفاوت هستند. این توابع ..... نامیده می‌شوند.

۶-۲ با توجه به برنامه شکل ۶-۴۰، قلمرو هر یک از عناصر زیر را تعیین کنید:

a) متغییر x در main.

b) متغیر y در cube.

c) تابع cube.

d) تابع main.

e) نمونه اولیه تابع برای cube.

f) شناسه y در نمونه اولیه تابع برای cube.

۶-۳ در برنامه زیر، قلمرو هر کدام یک از عناصر زیر را تعیین کنید:

a) متغیر **x**

b) متغیر **y**

c) تابع **cube**

d) تابع **paint**

e) متغیر **yPos**

۶-۴ برای هر کدامیک از موارد زیر یک سرآیند تابع ایجاد کنید:

a) تابع **hypotenuse** که دو آرگومان **side1** و **side2** از نوع **double** دریافت و نتیجه‌ای از نوع **double** برگشت دهد.

b) تابع **smallest** که سه آرگومان **x**، **y** و **z** دریافت و یک مقدار صحیح برگشت دهد.

c) تابع **instructions**، که هیچ آرگومانی دریافت نکرده و هیچ مقداری بر نمی‌گرداند.

d) تابع **intToSingle**، که یک آرگومان از نوع صحیح بنام **number** دریافت و مقداری از نوع **float** برگشت دهد.

پاسخ خودآزمایی

۶-۱ a) کلاس و تابع. b) فراخوانی. c) محلی. d) **return**. e) **void**. f) قلمرو. g) **return;**، **return** *عبارت* و **{**. h) **Random.Next** i) اتوماتیک. j) بازگشتی. k) پایه (اصلی). l) Overload. m) بلوک. n) تکرار. o) انتخاب. p) تابع. q) مشابه.

۶-۲ a) اشتباه. تابع **Abs** از کلاس **Math** مقدار مطلق یک عدد را باز می‌گرداند. b) صحیح. c) صحیح. d) صحیح. e) اشتباه. نوع **Char** می‌تواند به نوع **int** با تبدیل کاهشی، برگردانده شود. f) اشتباه. تابعی که بصورت بازگشتی خود را فراخوانی می‌کند با نام فراخوانی بازگشتی یا گام بازگشتی شناخته

می‌شود. g) صحیح. h) اشتباه. بازگشتی بی‌پایان زمانی رخ می‌دهد که تابع بازگشتی هرگز به حالت پایه نرسد. i) صحیح. j) صحیح.

۶-۳ a) قلمرو کلاس. b) قلمرو بلوک. c) قلمرو کلاس. d) قلمرو کلاس. e) قلمرو بلوک.

۶-۴

(a

```
double hypotenuse (double side1, double side2)
```

(b

```
int smallest (int x,
              int y, int z )
```

(c

```
void instructions( )
```

(d

```
float intToFloat (int number)
```

۶-۵

a) خطا: تابع **h** در تابع **g** تعریف شده است.

اصلاح: تعریف تابع **h** به خارج از تعریف تابع **g** منتقل شود.

b) خطا: فرض تابع بر برگشت یک مقدار **int** است، اما چنین نیست.

اصلاح: حذف عبارت **result = x + y** و جایگزین کردن عبارت زیر:

```
return x + y;
```

یا افزودن عبارت زیر به انتهای بدنه تابع:

```
return result;
```

c) خطا: نتیجه **n + sum(n-1)** توسط این تابع برگشت داده نمی‌شود.

اصلاح: عبارت موجود در شرط else بصورت زیر نوشته شود، **return n + sum(n-1);**

d) خطا: قرار دادن سیمکولن پس از پرانتز سمت راست و تعریف مجدد پارامتر a در تعریف تابع هر دو اشتباه است.

اصلاح: حذف سیمکولن و حذف اعلان **float a;**

e) خطا: تابع مقداری باز می‌گرداند که در نظر گرفته نشده است.

اصلاح: تغییر نوع برگشتی به **int**

تمرینات

۶-۶ در پارکینگی هزینه نگهداری هر اتومبیل تا سه ساعت حداقل ۶ دلار است. هزینه هر ساعت اضافی، ۵/۱ دلار علاوه بر هزینه سه ساعت می‌باشد. حداکثر هزینه نگهداری در ۲۴ ساعت معادل ۲۵ دلار است. فرض کنید که اتومبیل‌ها فقط می‌توانند تا ۲۴ ساعت در پارکینگ نگهداری شوند. برنامه‌ای بنویسید که هزینه نگهداری اتومبیل هر مشتری را محاسبه و به نمایش درآورد. باید ساعت ورود هر اتومبیل ثبت شود. برنامه باید از تابعی بنام **CalculateCharges** برای محاسبه هزینه هر ماشین استفاده کند.

۶-۷ کامپیوتر در امر آموزش نقش مهمی برعهده دارد. برنامه‌ای بنویسید که به دانش‌آموزان مدرسه ابتدائی، جدول ضرب آموزش دهد. از تابع **Next** برای ایجاد دو عدد مثبت یک رقمی استفاده کنید. سپس سئوالی مانند

```
How much is 6 times 7?
```

پرسیده شود. برنامه پاسخ وارد شده را چک کرده و در صورت صحیح بودن، پیغام "Very good" را چاپ کرده و سئوال دیگری مطرح کند. در غیر اینصورت پیغام "No.Please try again" چاپ شده و همان سئوال تا دریافت پاسخ صحیح تکرار شود.

۶-۷ تابعی بنویسید که یک عدد صحیح دریافت کرده و آنرا معکوس کند. برای مثال اگر عدد دریافتی 8456 باشد، خروجی تابع عدد 6548 را به نمایش درآورد.

۶-۸ به عددی، عدد اول گفته می‌شود که فقط به یک و خودش قابل تقسیم باشد برای مثال اعداد 2، 3 و 5 عدد اول هستند، اما اعداد 9، 8، 6 و 4 عدد اول نیستند.

a) تابعی بنویسید تا مشخص کند آیا عددی اول است یا خیر.

b) با استفاده از این تابع در برنامه، اعداد اول قرار گرفته از 1000 تا 1 را مشخص کرده و به نمایش درآورید.

فصل هفتم

آرایه‌ها و بردارها

اهداف

- **استفاده از ساختمان داده آرایه برای عرضه مجموعه‌ای از ایتم‌های داده مرتبط باهم.**
- **استفاده از آرایه برای ذخیره سازی، مرتب سازی و جستجوی لیست‌ها و جداول.**
- **اعلان آرایه، مقداردهی اولیه آرایه‌ها و مراجعه به عناصر مختلف آرایه.**
- **ارسال آرایه‌ها به توابع.**
- **تکنیک‌های اولیه جستجو و مرتب سازی.**
- **اعلان و کار با آرایه‌های چندبعدی.**
- **استفاده از الگوی vector از کتابخانه استاندارد C++.**

رئوس مطالب

۷-۱ مقدمه

در این فصل به معرفی یکی از مباحث مهم در ساختمان‌های داده‌ها می‌پردازیم، کلکسیونی از ایتم‌های داده مرتبط باهم. آرایه‌ها، ساختمان‌های داده متشکل از ایتم‌های داده مرتبط به‌هم و از یک نوع هستند. از فصل سوم با کلاس‌ها آشنا هستید. در فصل ۹، در ارتباط با نظریه ساختمان بحث خواهیم کرد. ساختمان‌ها و کلاس هر دو قادر به نگهداری ایتم‌های داده مرتبط هستند که این داده‌ها می‌توانند از نوع‌های مختلف باشند. آرایه‌ها، ساختمان‌ها و کلاس‌ها از موجودیت‌های «استاتیک» محسوب می‌شوند که در اینحالت سایز آنها در مدت زمان اجرای برنامه ثابت باقی می‌ماند. البته می‌توان به کمک کلاس ذخیره سازی اتوماتیک در این روش اعمال نفوذ کرد.

پس از بحث در مورد نحوه اعلان، ایجاد و مقداردهی اولیه آرایه‌ها، در این فصل به بررسی چندین مثال کاربردی خواهیم پرداخت که نحوه کار با آرایه‌ها را نشان می‌دهند. سپس به توضیح نحوه ارائه رشته‌های کاراکتری توسط آرایه‌های کاراکتری می‌پردازیم. مثالی در ارتباط با جستجوی آرایه‌ها به منظور یافتن عناصر خاصی در یک آرایه مطرح می‌کنیم. همچنین در این فصل به معرفی یکی از مهمترین برنامه‌های کاربردی در علم کامپیوتر می‌پردازیم، که مرتب‌سازی داده‌ها می‌باشد. دو بخش از این فصل اختصاص به مبحث آموزشی کلاس **GradeBook** مطرح شده در فصل‌های ۶ الی۳ دارد. در واقع، از آرایه‌ها به نحوی استفاده شده تا کلاس قادر به نگهداری مجموعه‌ای از نمرات در حافظه شده و بتواند این نمرات را تجزیه و تحلیل نماید، دو قابلیتی که در کلاس **GradeBook** نسخه‌های قبلی وجود نداشت. این مثال‌ها و مثال‌های

دیگر این فصل به توضیح روشی می‌پردازند که در آن آرایه‌ها به برنامه‌نویس امکان سازماندهی و کنترل بر روی داده‌ها را می‌دهند.

سبک آرایه که در سرتاسر این فصل از آن استفاده کرده‌ایم، سبک آرایه‌های مبتنی بر اشاره‌گر در C است. در فصل هشتم با اشاره‌گرها آشنا خواهید شد. بخش پایانی این فصل در ارتباط با شی‌هایی بنام *بردار* **(vector)** است که تکامل یافته آرایه‌ها می‌باشد. متوجه خواهید شد که این آرایه‌ها مبتنی بر شی به نسبت آرایه‌های مبتنی بر اشاره‌گر سبک C ایمن‌تر و تطبیق‌پذیرتر هستند.

۷-۲ آرایه‌ها

یک آرایه گروهی از مکان‌های حافظهٔ پشت سرهم هم نوع می‌باشند. برای اشاره به یک مکان مشخص یا عنصری در یک آرایه، نام آرایه و سپس شماره مکان یا موقعیت عنصر مورد نظر آورده می‌شود.

در شکل ۱-۷ یک آرایه از نوع صحیح با نام **c** نشان داده شده است. این آرایه حاوی 12 عنصر است که هر کدام را می‌توان با بکار بردن نام آرایه و شمارهٔ موقعیت در درون یک جفت براکت**[]** مورد مراجعه قرار داد. اولین عنصر در هر آرایه، عنصر صفر نامیده می‌شود، از اینرو اولین عنصر در آرایه **c**، بصورت **c[0]**، دومین عنصر بصورت **c[1]**، و هفتمین عنصر بصورت **c[6]** و الی آخر مورد مراجعه قرار می‌گیرد. در حالت کلی عنصر iام در آرایه **c** بصورت **c[i – 1]** مورد مراجعه قرار می‌گیرد. به عددی که در درون براکت‌ها آورده می‌شود، شاخص (یا ساب‌اسکریپت) گفته می‌شود. شاخص باید یک عدد صحیح یا عبارت صحیحی باشد. اگر برنامه‌ای از یک عبارت بعنوان شاخص استفاده کند، ابتدا این عبارت برای تعیین مقدار شاخص ارزیابی می‌شود. برای مثال، اگر متغیر **a** معادل **5** باشد و متغیر **b** معادل **6**، عبارت

```
c[a + b] += 2;
```

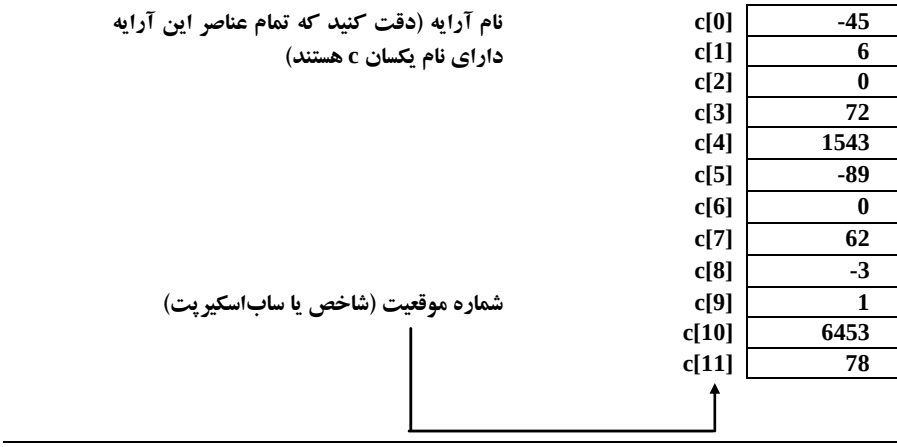
اقدام به افزودن عدد **2** به عنصر یازدهم آرایه **c[11]** خواهد کرد. اجازه دهید تا از نزدیک به بررسی آرایه **c** در شکل ۱-۷ بپردازیم. نام کل آرایه **c** است. به 12 عنصر این آرایه بصورت **c[0]** تا **c[11]** می‌توان دسترسی پیدا کرد. مقدار **c[0]** برابر **45-**، مقدار **c[1]** برابر **6**، مقدار **c[2]** برابر **0**، مقدار **c[7]** برابر **62** و مقدار **c[11]** برابر **78** است. مقادیر ذخیره شده در آرایه‌ها از قابلیت بکارگیری در محاسبات برنامه‌های مختلف دارا هستند.

برای مثال، برای چاپ مجموع مقادیر در سه عنصر ابتدایی آرایه **c** می‌توانیم از عبارت زیر استفاده کنیم:

```
cout << c[0] + c[1] + c[2] << endl;
```

برای تقسیم مقدار، هفتمین عنصر آرایه **c** به **2** و تخصیص نتیجه به متغیر **x**، می‌توانیم از عبارت زیر استفاده کنیم:

```
x = c[6] / 2;
```



شکل ۱-۷ آرایه‌ای حاوی 12 عنصر.

خطای برنامه‌نویسی

توجه به تفاوت موجود مابین «هفتمین عنصر آرایه» و «عنصر هفتم آرایه» مهم است. شاخص آرایه‌ها با صفر شود می‌شود، از اینرو «هفتمین عنصر آرایه» دارای شاخص 6 است در حالیکه «عنصر هفتم آرایه» دارای شاخص 7 می‌باشد و در واقع هشتمین عنصر آرایه است. متاسفانه این تفاوت غالباً موجب رخ دادن خطای ***off-by-one*** *می‌شود. برای اجتناب از چنین خطایی، به عناصر آرایه بطور صریح و با نام آرایه و شماره شاخص مراجعه می‌کنیم (برای مثال c [6] یا c [7]).*

در واقع براکت‌های در برگیرنده شاخص یک آرایه، یک عملگر در C++ هستند. براکت‌ها تقدم یکسان با پرانتزها دارند. در جدول شکل ۲-۷ الویت و تقدم عملگرهای معرفی شده تا بدین مرحله آورده شده است. در این جدول نمایش اولویت‌ها از بالا به پایین است.

عملگر	ارتباط	نوع
() []	چپ به راست	حداکثر
++ -- ! static_cast<type>(operand)	چپ به راست	غیربایتری
++ -- + - !	راست به چپ	
* / %	چپ به راست	تعددی
+ -	چپ به راست	افزاینده کاهنده
<< >>	چپ به راست	درج/استخراج
< <= > >=	چپ به راست	رابطه‌ای
== !=	چپ به راست	برابری

عملگر	ارتباط	نوع
&&	چپ به راست	AND منطقی
||	چپ به راست	OR منطقی
?:	راست به چپ	شرطی
= += -= *= /= %=	راست به چپ	تخصیصی
,	چپ به راست	کاما

شکل ۲-۷ | تقدم و الویت عملگرها.

۳-۷ اعلان آرایه‌ها

آرایه‌ها اشغالگر فضای حافظه هستند. برنامه‌نویس تعیین کننده نوع عناصر و تعداد آنها بصورت زیر است:

***نوع* نام آرایه [سایز آرایه] ;**

و کامپایلر میزان فضایی مورد نیاز برای آرایه را رزرو می‌کند. «سایز آرایه» باید یک عدد صحیح بزرگتر از صفر باشد. برای مثال، برای اینکه کامپایلر 12 عنصر برای یک آرایه صحیح بنام **c** رزرو کند، از اعلان زیر استفاده می‌کنیم.

```
int c[12]; // c is an array of 12 integers
```

می‌توان با یک اعلان برای چندین آرایه، حافظه رزرو کرد. در اعلان زیر مبادرت به رزرو 100 عنصر برای آرایه صحیح **b** و 27 عنصر برای آرایه صحیح **x** شده است.

```
int b[100], // b is an array of 100 integers
    x[27]; //x is an array of 27 integers
```

برنامه‌نویسی ایده‌ال

به منظور افزایش خوانایی، اصلاح پذیری آسانتر و نوشتن راحت توضیحات، ترجیح می‌دهیم که یک آرایه در هر اعلان، اعلان گردد.

آرایه‌ها قادر به نگهداری مقادیری هستند که از آن نوع اعلان شده‌اند. برای مثال، یک آرایه از نوع **char** می‌تواند برای ذخیره‌سازی یک رشته کاراکتری بکار گرفته شود. تا بدین مرحله، از شی‌های **string** برای ذخیره سازی رشته‌های کاراکتری استفاده کرده‌ایم. در بخش ۴-۷ به معرفی نحوه استفاده از آرایه‌های کاراکتری برای ذخیره سازی رشته‌ها خواهیم پرداخت.

٤-۷ مثال‌هایی از کاربرد آرایه

در این بخش به ارائه چندین مثال می‌پردازیم که نحوه اعلان، تخصیص و مقداردهی آرایه‌ها و همچنین کار با عناصر آرایه‌ها را نشان می‌دهد.

اعلان آرایه و استفاده از یک حلقه برای مقداردهی اولیه عناصر آرایه

در برنامه شکل ۳-۷ مبادرت به اعلان آرایه صحیح **n** با 10 عنصر شده است (خط 12). در خطوط 15-16 از یک عبارت **for** برای مقداردهی اولیه عناصر آرایه با صفر استفاده شده است. اولین عبارت خروجی

(خط 18) نشان‌دهنده سرآیندهای ستون است که عبارت **for** موجود در خطوط 22-21 عناصر و مقادیر آرایه را با فرمت جدولی در زیر آنها چاپ می‌کند. بخاطر دارید که **setw** تصریح کننده طول میدان است.

```
1  // Fig. 7.3: fig07_03.cpp
2  // Initializing an array.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <iomanip>
8  using std::setw;
9
10 int main()
11 {
12    int n[ 10 ]; // n is an array of 10 integers
13
14    // initialize elements of array n to 0
15    for ( int i = 0; i < 10; i++ )
16       n[ i ] = 0; // set element at location i to 0
17
18    cout << "Element" << setw( 13 ) << "Value" << endl;
19
20    // output each array element's value
21    for ( int j = 0; j < 10; j++ )
22       cout << setw( 7 ) << j << setw( 13 ) << n[ j ] << endl;
23
24    return 0; // indicates successful termination
25 } // end main
```

```
Element             Value
      0                 0
      1                 0
      2                 0
      3                 0
      4                 0
      5                 0
      6                 0
      7                 0
      8                 0
      9                 0
```

شکل ۳-۷ | مقداردهی اولیه عناصر آرایه با صفر و چاپ آرایه.

مقداردهی اولیه آرایه در زمان اعلان با لیست مقداردهی

می‌توان عناصر آرایه را در زمان اعلان آرایه و با قرار دادن نماد مساوی پس از نام آرایه و یک لیست جدا شده با کاما (قرار گرفته در میدان اکولادها، { }) مقداردهی اولیه کرد. در برنامه شکل ۴-۷ از یک لیست مقداردهی اولیه برای مقداردهی یک آرایه صحیح با 10 مقدار (خط 13) و چاپ آرایه با فرمت جدولی (خطوط 19-15) استفاده شده است.

اگر مقداردهی اولیه کمتر از تعداد عناصر آرایه باشد، مابقی عناصر آرایه با صفر مقداردهی خواهند شد. برای مثال، عناصر آرایه n در شکل ۳-۷ را می‌توان با عبارت زیر، تماماً با صفر مقداردهی اولیه کرد

```
int n[10] = {0}; // intialize elements of array n to 0
```

در این اعلان بصورت صریح اولین عنصر با صفر مقداردهی اولیه شده و 9 عنصر باقیمانده بصورت ضمنی با صفر مقداردهی اولیه می‌شوند، چرا که تعداد مقداردهی اولیه کمتر از تعداد عناصر آرایه است.

آرایه‌های اتوماتیک بصورت ضمنی با صفر مقداردهی اولیه نمی‌شوند، در حالیکه آرایه‌های استاتیک می‌توانند چنین کاری کنند. برنامه نویس بایستی حداقل اولین عنصر آرایه را در لیست مقداردهی اولیه، با صفر مقداردهی کند تا مابقی عناصر باقیمانده از آرایه بصورت ضمنی با صفر مقداردهی اولیه شوند. روش مقداردهی اولیه عرضه شده در برنامه ۳-۷ در هر بار اجرای برنامه بکار گرفته می‌شود.

اگر سایز آرایه به هنگام اعلان به همراه یک لیست مقداردهی اولیه، از قلم بیفتد، کامپایلر برحسب تعداد عناصر موجود در لیست مقداردهی اولیه، مبادرت به تعیین تعداد عناصر آرایه می‌کند. برای مثال،

```
int n[] = {1 , 2 , 3 , 4 , 5};
```

یک آرایه با پنج عنصر بوجود می‌آورد.

اگر سایز آرایه و لیست مقداردهی اولیه در اعلان یک آرایه مشخص شده باشند، بایستی تعداد عناصر موجود در لیست مقداردهی اولیه کمتر یا برابر با سایز آرایه باشد. برای مثال در اعلان آرایه زیر

```
int n[5] = {32 , 27 , 64 , 18 , 95 , 14};
```

با خطای کامپایل مواجه خواهید شد، چرا که لیست مقداردهی اولیه دارای شش عنصر است در حالیکه آرایه فقط پنج عنصر دارد.

خطای برنامه‌نویسی

تدارک دیدن عناصر بیشتر در لیست مقداردهی اولیه، به نسبت سایز آرایه، خطای کامپایل است.

خطای برنامه‌نویسی

فراموش کردن مقداردهی اولیه عناصر یک آرایه که باید مقداردهی اولیه شوند، یک خطای منطقی است.

```
1  // Fig. 7.4: fig07_04.cpp
2  // Initializing an array in a declaration.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <iomanip>
8  using std::setw;
9
10 int main()
11 {
12    // use initializer list to initialize array n
13    int n[ 10 ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
14
15    cout << "Element" << setw( 13 ) << "Value" << endl;
16
17    // output each array element's value
18    for ( int i = 0; i < 10; i++ )
19       cout << setw( 7 ) << i << setw( 13 ) << n[ i ] << endl;
20
21    return 0; // indicates successful termination
22 } // end main
```

```
Element         Value
      0            32
      1            27
      2            64
      3            18
      4            95
      5            14
      6            90
```

```
      7              70
      8              60
      9              37
```

شکل ٤-٧ | مقداردهی اولیه عناصر آرایه به هنگام اعلان.

تعیین سای‌ز آرایه با متغیر ثابت و تنظیم عناصر آرایه از طریق محاسبه

در برنامه شکل ۵-۷ عناصر یک آرایه ده عنصری بنام s با مقادیر زوج 2,4,6,...,20 تنظیم شده (خطوط 17-18) و آرایه با فرمت جدولی چاپ شده است (خطوط 20-24). این اعداد با ضرب هر مقدار پی‌درپی شمارنده حلقه در 2 و جمع آن با 2 تولید می‌شوند (خط 18).

```
1  // Fig. 7.5: fig07_05.cpp
2  // Set array s to the even integers from 2 to 20.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <iomanip>
8  using std::setw;
9
10 int main()
11 {
12    // constant variable can be used to specify array size
13    const int arraySize = 10; // must initialize in declaration
14
15    int s[ arraySize ]; // array s has 10 elements
16
17    for ( int i = 0; i < arraySize; i++ ) // set the values
18       s[ i ] = 2 + 2 * i;
19
20    cout << "Element" << setw( 13 ) << "Value" << endl;
21
22    // output contents of array s in tabular format
23    for ( int j = 0; j < arraySize; j++ )
24       cout << setw( 7 ) << j << setw( 13 ) << s[ j ] << endl;
25
26    return 0; // indicates successful termination
27 } // end main
```

```
Element             Value
      0                 2
      1                 4
      2                 6
      3                 8
      4                10
      5                12
      6                14
      7                16
      8                18
      9                20
```

شکل ۵-۷ |تولید مقادیر برای وارد کردن به آرایه.

در خط 13 از یک توصیف کننده **const** برای اعلان یک متغیر ثابت بنام **arraySize** با مقدار 10 استفاده شده است. بایستی متغیرهای ثابت به هنگام اعلان مقداردهی شوند. پس از آن مقدار این متغیرها قابل تغییر نیست (همانند برنامه ۶-۷ و ۷-۷). به متغیرهای ثابت، ثابت‌های نامی یا متغیرهای فقط خواندنی هم گفته می‌شود.

خطای برنامه‌نویسی

در صورت تخصیص یک مقدار به یک متغیر ثابت پس از اعلان آن، با خطای کامپایل مواجه خواهید شد.

خطای برنامه‌نویسی

نتیجه تخصیص یک مقدار به یک متغیر ثابت در یک عبارت اجرایی، خطای کامپایل است.

می‌توان متغیرهای ثابت را در هر کجای که یک عبارت ثابت مورد نیاز است بکارگرفت. در برنامه شکل ۵-۷ متغیر ثابت **arraySize** سایز آرایه **s** را در خط 15 تصریح کرده است.

خطای برنامه‌نویسی

فقط می‌توان از ثابت‌ها برای اعلان سایز آرایه‌های اتوماتیک و استاتیک استفاده کرد. عدم استفاده از یک ثابت به این منظور، خطای کامپایل بدنبال خواهد داشت.

با استفاده از متغیرهای ثابت در تعیین سایز آرایه، خوانایی برنامه افزایش پیدا می‌کند. در برنامه شکل ۵-۷ اولین ساختار **for** یک آرایه 100 عنصری را با تغییر ساده مقدار **arraySize** در اعلان خود از 10 تا 1000 پر می‌کند. اگر متغیر ثابت **arraySize** بکار گرفته نشده بود، مجبور بودیم تا خطوط 15، 17 و 23 برنامه را برای کار با آرایه 1000 عنصری تغییر دهیم. همانطوری که برنامه‌ها بزرگتر می‌شوند، این تکنیک می‌تواند در نوشتن برنامه‌های واضح‌تر و اصلاح‌پذیرتر بکار گرفته شود.

مهندسی نرم‌افزار

تعریف سایز هر آرایه بصورت یک متغیر ثابت بجای یک ثابت لیترال می‌تواند در ایجاد برنامه‌های بسط‌پذیر موثر باشد.

برنامه‌نویسی ایده‌ال

تعریف سایز آرایه بصورت یک متغیر ثابت بجای یک ثابت لیترال، سبب ایجاد برنامه‌های واضح‌تر می‌شود. این تکنیک سبب حذف اعداد جادویی می‌گردد.

```
1  // Fig. 7.6: fig07_06.cpp
2  // Using a properly initialized constant variable.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     const int x = 7; // initialized constant variable
10
11    cout << "The value of constant variable x is: " << x << endl;
12
13    return 0; // indicates successful termination
14 } // end main
```

```
The value of constant variable x is:7
```

شکل ٦-٧ |مقداردهی اولیه و استفاده از یک متغیر ثابت.

```
1  // Fig. 7.7: fig07_07.cpp
2  // A const variable must be initialized.
3
4  int main()
```

```
5  {
6     const int x; // Error: x must be initialized
7
8     x = 7; // Error: cannot modify a const variable
9
10    return 0; // indicates successful termination
11 } // end main
```

Borland C++ command-line compiler error message:

```
Error E2304 fig07_07.cpp 6: Constant variablr 'x' must be initialized
   in function main()
Error E2304 fig07_07.cpp 8: Cannot modify a const object in function
main()
```

Microsoft Visual C++ .NET compiler error message:

```
C:\cpphtp5e_examples\ch07\fig07_07.cpp(6):error C2734:'x': const object
   must be initialized if not extern
C:\cpphtp5e_examples\ch07\fig07_07.cpp(8):error C2166: l-value specifies
   const object
```

GNU C++ compiler error message:

```
fig07_07.cpp:6: error: unitialized const 'x'
fig07_07.cpp:8: error: assignment of read-only variable 'x'
```

شکل ۷–۷ |متغیر ثابت بایستی مقداردهی اولیه شود.

بدست آوردن مجموع عناصر آرایه

غالباً عناصر یک آرایه نشاندهنده دنباله‌ای از مقادیر هستند که در محاسبات بکار گرفته می‌شوند. برای مثال، اگر عناصر یک آرایه نشاندهنده نمرات تعدادی از دانشجویان باشد، ممکن است استاد علاقمند به دانستن میانگین نمرات کلاس این عده از دانشجویان باشد. در این مثال از کلاس **GradeBook** در برنامه‌های شکل ۱۶–۷ و ۱۷–۷ استفاده شده است. همچنین در برنامه‌های شکل ۲۳–۷ و ۲۴–۷ از این تکنیک استفاده کرده‌ایم. در برنامه شکل ۸–۷ آرایه **a** با ده عنصر اعلان، تخصیص و مقداردهی اولیه شده است (خط 10).

```
1  // Fig. 7.8: fig07_08.cpp
2  // Compute the sum of the elements of the array.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     const int arraySize = 10; // constant variable indicating size of array
10    int a[ arraySize ] = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
11    int total = 0;
12
13    // sum contents of array a
14    for ( int i = 0; i < arraySize; i++ )
15       total += a[ i ];
16
17    cout << "Total of array elements: " << total << endl;
18
19    return 0; // indicates successful termination
20 } // end main
```

```
Total of array elements: 849
```

شکل ۸–۷ | محاسبه مجموع عناصر موجود در یک آرایه.

خطوط 15-14 در بدنه ساختار **for**، عمل جمع را انجام می‌دهد. بصورت جایگزین می‌توان مقادیر تدارک دیده شده بعنوان مقادیر اولیه برای آرایه **a** را از طریق کاربر یا یک فایل وارد برنامه ساخت. برای کسب اطلاعات بیشتر در زمینه وارد کردن مقادیر به برنامه‌ها می‌توانید به فصل ۱۷ مراجعه کنید. برای مثال، عبارت **for**

```
for ( int j = 0; j < arraySize; j++ )
   cin >> a [ j ];
```

در هر بار یک مقدار از صفحه‌کلید خوانده و آنرا در عنصر **a[j]** ذخیره می‌سازد.

نمایش گرافیکی داده‌های آرایه توسط نمودارهای میله‌ای

بسیاری از برنامه‌ها داده‌های خود را با فرمت‌های گرافیکی یا بصری به اطلاع کاربران خود می‌رسانند. برای مثال، غالباً مقادیر عددی بصورت میله‌های در یک نمودار میله‌ای به نمایش در می‌آیند که میله‌های بلندتر نشاندهنده مقادیر عددی بزرگتر هستند. یکی از ساده‌ترین روش‌های نمایش گرافیکی داده‌های عددی استفاده از یک نمودار میله‌ای است که هر مقدار عددی را بصورت میله‌ای از ستاره‌ها (*) به نمایش در می‌آورد.

غالباً اساتید علاقمند به بررسی توزیع نمرات در یک امتحان یا آزمون هستند. فرض کنید که نمرات عبارتند از 87 ، 68 ، 94 ، 100 ، 83 ، 78 ، 85 ، 91 ، 76 و 87 باشند. توجه کنید که در اینجا فقط یک نمره 100، دو نمره در محدودهٔ 90، چهار نمره در محدودهٔ 80، دو نمودار محدودهٔ 70 و یک نمره در محدودهٔ 60 قرار دارد و هیچ نمره‌ای پایین‌تر از 60 وجود ندارد. در مثال بعدی (شکل ۹-۷) این نمرات در یک آرایه 11 عنصری ذخیره شده که هر یک متناظر با یک رده از نمرات می‌باشند. برای مثال، **n[0]** نشاندهنده تعداد نمرات در محدودهٔ 9-0، **n[7]** نشاندهنده تعداد نمرات در محدودهٔ 79-70 است و **n[10]** نشاندهنده تعداد نمره‌های 100 است. دو نسخه از کلاس **GradeBook** (شکل‌های ۱۶-۷ و ۱۷-۷ و شکل‌های ۲۳-۷ و ۲۴-۷) حاوی کدی هستند که دفعات تکرار این نمرات را محاسبه می‌کنند. در این مرحله، بصورت غیراتوماتیک آرایه‌ای را با مجموعه‌ای از نمرات ایجاد می‌کنیم.

```
1  // Fig. 7.9: fig07_09.cpp
2  // Bar chart printing program.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <iomanip>
8  using std::setw;
9
10 int main()
11 {
12    const int arraySize = 11;
13    int n[ arraySize ] = { 0, 0, 0, 0, 0, 0, 1, 2, 4, 2, 1 };
14
15    cout << "Grade distribution:" << endl;
```

```
16
17    // for each element of array n, output a bar of the chart
18    for ( int i = 0; i < arraySize; i++ )
19    {
20       // output bar labels ("0-9:", ..., "90-99:", "100:" )
21       if ( i == 0 )
22          cout << "  0-9: ";
23       else if ( i == 10 )
24          cout << "  100: ";
25       else
26          cout << i * 10 << "-" << ( i * 10 ) + 9 << ": ";
27
28       // print bar of asterisks
29       for ( int stars = 0; stars < n[ i ]; stars++ )
30          cout << '*';
31
32       cout << endl; // start a new line of output
33    } // end outer for
34
35    return 0; // indicates successful termination
36 } // end main
```

```
Grade distribution :
  0-9:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: *
70-79: **
80-89: ****
90-99: **
  100: *
```

شکل ۹-۷ | برنامه چاپ نمودار میله‌ای.

برنامه اعداد را از آرایه خوانده و اطلاعات را بصورت نمودار میله‌ای به نمایش در می‌آورد. برنامه طول هر گراف را با میله‌ای از ستارهٔ که نشاندهنده تعداد نمرات در آن محدودهٔ هستند، نشان می‌دهد. برای قرار دادن یک برچسب (عنوان) برای هر میله، خطوط 26-21 محدودهٔ هر نمره را (مثلاً ":79-70") برحسب مقدار جاری متغیر شمارنده **i** چاپ می‌کنند. عبارت **for** تودرتو (خطوط 30-29) میله‌ها را چاپ می‌کند. به شرط تکرار حلقه در خط 29 دقت کنید (**stars<n[i]**). هر بار که برنامه به **for** داخلی می‌رسد، شمارش حلقه از 0 تا **n[i]** صورت می‌گیرد، از اینرو با استفاده از این مقدار در آرایه **n** تعداد ستاره‌های که باید به نمایش درآیند، مشخص می‌شود. در این مثال، حاصل **n[0]-n[5]** صفر است، چرا که هیچ دانشجویی نمره‌ای کمتر از 60 دریافت نکرده است. بنابر این برنامه در کنار شش محدودهٔ امتیاز اول هیچ ستاره‌ای به نمایش در نیاورده است.

خطای برنامه‌نویسی

*اگرچه امکان استفاده از یک متغیر کنترلی یکسان در یک عبارت **for** و عبارت **for** دوم قرار گرفته در درون اولی وجود دارد، اما بدلیل اجتناب از رخ دادن خطاهای منطقی از آن اجتناب کرده‌ایم.*

استفاده از عناصر آرایه بعنوان شمارنده

گاهی اوقات، برنامه‌ها از متغیرهای شمارنده برای تحلیل داده‌ها استفاده می‌کنند. در برنامه ۹-۶ از شمارنده‌های مختلف در برنامه پرتاب طاس برای ردگیری تعداد رخ دادهای هر وجه طاس استفاده کردیم. نسخه آرایه‌ای این برنامه در شکل ۱۰-۷ آورده شده است.

در برنامه ۱۰-۷ از آرایه **frequency** برای شمارش رخ دادهای هر وجه طاس استفاده شده است (خط 20). یک عبارت در خط 26 این برنامه جایگزین ساختار **switch** در خطوط 52-30 از برنامه ۹-۶ شده است. در خط 26 از یک مقدار تصادفی برای تعیین اینکه کدام عنصر **frequency** در زمان تکرار هر حلقه افزایش یافته، استفاده شده است. محاسبه بکار رفته در خط 26 یک شاخص تصادفی از 1 تا 6 ایجاد می‌کند، از اینرو آرایه **frequency** بایستی بقدر کافی برای نگهداری شش شمارنده بزرگ باشد. با این همه، ما از یک آرایه هفت عنصری استفاده کرده‌ایم تا **frequency[0]** را در نظر نگیریم. در اینحالت بسیار منطقی خواهد بود که برای وجه 1 طاس مقدار **frequency[1]** بجای **frequency[0]** افزایش یابد. بنابر این از مقدار هر وجه بعنوان شاخص آرایه **frequency** استفاده شده است. همچنین خطوط 61-56 از برنامه ۹-۶ را با حلقه‌ای که در میان آرایه **frequency** برای چاپ نتایج حرکت می‌کند، جایگزین کرده‌ایم (خطوط 33-31).

```
1  // Fig. 7.10: fig07_10.cpp
2  // Roll a six-sided die 6,000,000 times.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <iomanip>
8  using std::setw;
9
10 #include <cstdlib>
11 using std::rand;
12 using std::srand;
13
14 #include <ctime>
15 using std::time;
16
17 int main()
18 {
19    const int arraySize = 7; // ignore element zero
20    int frequency[ arraySize ] = { 0 };
21
22    srand( time( 0 ) ); // seed random number generator.
23
24    // roll die 6,000,000 times; use die value as frequency index
25    for ( int roll = 1; roll <= 6000000; roll++ )
26       frequency[ 1 + rand() % 6 ]++;
27
28    cout << "Face" << setw( 13 ) << "Frequency" << endl;
29
30    // output each array element's value
31    for ( int face = 1; face < arraySize; face++ )
32       cout << setw( 4 ) << face << setw( 13 ) << frequency[ face ]
33          << endl;
34
35    return 0; // indicates successful termination
36 } // end main
```

Face	Frequency
1	1000167
2	1000149
3	1000152
4	998748
5	999626
6	1001158

شکل ۱۰-۷ | برنامه پرتاپ طاس با استفاده از آرایه بجایswitch.

استفاده از آرایه‌ها برای تحلیل نتایج

در مثال بخش قبلی از آرایه برای بررسی اطلاعات جمع‌آوری شده از یک امتحان استفاده شده بود حال به مسئله زیر توجه کنید:

از چهل دانشجو در مورد کیفیت غذای عرضه شده در رستوران دانشگاه سئوال شده و پاسخ دانشجویان می‌تواند در محدودهٔ 10 تا 1 قرار داشته باشد. به اینصورت که 1 نشاندهنده کیفیت بسیار پایین و 10 کیفیت عالی است. پاسخ چهل دانشجو را در یک آرایه قرار داده و میزان و تعداد پاسخ‌های همسان را مشخص سازید.

این مسئله با استفاده از یک آرایه در برنامه شکل ۱۰-۷ ارائه شده است. در این برنامه علاقمند هستیم تا تعداد پاسخ‌های مطرح شده و نوع آنها را دسته‌بندی نمائیم. آرایه **responses** (خطوط 19-17) یک آرایه 40 عنصری از نوع صحیح و حاوی پاسخ‌های دانشجویان است. دقت کنید که این آرایه بصورت **const** اعلان شده است، و از اینرو مقادیر آن قابل تغییر نمی‌باشند (و نباید تغییر داده شوند). با استفاده از آرایه **frequency**، با 11 عنصر، می‌توانیم تعداد پاسخ‌های همسان را شمارش کنیم (خط 22). اولین عنصر آرایه، **frequency[0]** را نادیده گرفته‌ایم چراکه بسیار منطقی است که پاسخ **1** در **frequency[1]** قرار داده شود. می‌توانیم هر پاسخ را بصورت مستقیم بعنوان یک شاخص بر روی آرایه **frequency** بکار گیریم.

```
1  // Fig. 7.11: fig07_11.cpp
2  // Student poll program.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <iomanip>
8  using std::setw;
9
10 int main()
11 {
12    // define array sizes
13    const int responseSize = 40; // size of array responses
14    const int frequencySize = 11; // size of array frequency
15
16    // place survey responses in array responses
17    const int responses[ responseSize ] = { 1, 2, 6, 4, 8, 5, 9, 7, 8,
18       10, 1, 6, 3, 8, 6, 10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7,
19       5, 6, 6, 5, 6, 7, 5, 6, 4, 8, 6, 8, 10 };
```

```
20
21    // initialize frequency counters to 0
22    int frequency[ frequencySize ] = { 0 };
23
24    // for each answer, select responses element and use that value
25    // as frequency subscript to determine element to increment
26    for ( int answer = 0; answer < responseSize; answer++ )
27       frequency[ responses[ answer ] ]++;
28
29    cout << "Rating" << setw( 17 ) << "Frequency" << endl;
30
31    // output each array element's value
32    for ( int rating = 1; rating < frequencySize; rating++ )
33       cout << setw( 6 ) << rating << setw( 17 ) << frequency[ rating ]
34          << endl;
35
36    return 0; // indicates successful termination
37 } // end main
```

Rating	Frequency
1	2
2	2
3	2
4	2
5	5
6	11
7	5
8	7
9	1
10	3

شکل ۱۱-۷ | برنامه تحلیل پاسخ دانشجویان.

کارائی

گاهی اوقات توجه به کارایی سبب کاهش توجه به وضوح برنامه می‌شود.

حلقه **for** (خطوط 27-26) یک به یک پاسخ‌ها را از آرایه **responses** خوانده و یک واحد به ده شمارندهٔ در آرایه **frequency** اضافه می‌کند (از **frequency[1]** تا **frequency[10]**). عبارت کلیدی در خط 27 حلقه متبلور است، این عبارت به شمارنده مقتضی **frequency** که توسط مقدار **responses[answer]** تعیین می‌شود، یک واحد اضافه می‌کند.

اجازه دهید تا به بررسی چند تکرار ساختار **for** بپردازیم. هنگامی شمارنده **answer** برابر **0** است، **responses[answer]** مقدار **responses[0]** را خواهد داشت (مقدار1 در خط 17). از اینرو، در واقع عبارت **++frequancy[responses[answer]]** بصورت **++frequency[1]** تفسیر خواهد شد، به این معنی که اولین شمارنده در آرایه **frequency** یک واحد افزایش می‌یابد. در ارزیابی عبارت کار با ارزیابی مقدار داخلی‌ترین براکت‌های آغاز می‌شود. مقدار **answer** وارد عبارت شده و به ارزیابی براکت‌های بعدی پرداخته می‌شود(**responses[answer]**)، که مقدار مورد استفاده بعنوان شاخص برای آرایه **frequnecy** است و تعیین می‌کند کدام شمارنده باید افزایش یابد (در این مورد شمارنده **1**).

هنگامی که **answer** برابر **1** است، عبارت **responses[answer]** مقدار دومین عنصر را خواهد داشت (مقدار **2**). در نتیجه، عبارت

```
frequancy[responses[answer]]++
```

بصورت **++frequency[2]** تفسیر شده و موجب می‌شود تا عنصر **2** آرایه (سومین عنصر در آرایه) افزایش یابد. زمانیکه **answer** برابر **2** است، عبارت **frequency[answer]** (مقدار **6**) را خواهد داشت، از اینرو

```
frequancy[responses[answer]]++
```

بصورت **++frequency[6]** تفسیر شده و موجب می‌شود تا عنصر **6** آرایه (هفتمین عنصر در آرایه) افزایش یابد. دقت کنید که علیرغم تعداد پاسخ‌های مطرح شده، فقط به 11 عنصر آرایه برای تحلیل نتایج نیاز است چرا که تمام پاسخ‌ها در محدودهٔ مقادیر 10 تا 1 قرار دارند و مقادیر شاخص برای 11 عنصر آرایه 11 تا 0 هستند.

اگر داده‌ای حاوی مقدار خارج از محدودهٔ نظیر 13 باشد، برنامه مبادرت به افزودن **1** به **frequency[13]** خواهد کرد، که در اینحالت از مرزهای آرایه خارج خواهد شد. در زبان C++، چنین مراجعه‌ای توسط کامپایلر و در زمان اجرا مجاز شناخته می‌شود. در چنین وضعیتی برنامه از مرز آرایه عبور کرده و داده را در مکانی از حافظه ذخیره می‌کند، این عمل می‌تواند در مقدار متغیر دیگری در برنامه تغییر ایجاد کند و در پاسخ برنامه اشکال بوجود آورد.

خطای برنامه‌نویسی

مراجعه به یک عنصر خارج از مرزهای یک آرایه، خطای زمان اجرا بدنبال خواهد داشت.

اجتناب از خطا

زمانیکه حلقه‌ای در درون یک آرایه اجرا می‌شود، باید شاخص آرایه در بین صفر و مرز بالایی آرایه بماند. مقادیر اولیه و پایانی بکار رفته در ساختار تکرار باید از دسترسی به عناصری خارج از مرزهای آرایه اجتناب کنند.

C++ یک زبان بسط پذیر است. در بخش۱۱-۷ به معرفی کلاس vector (بردار) خواهیم پرداخت که به برنامه نویسان امکان انجام فرآیندهای را می‌دهد که انجام آنها در آرایه‌های توکار C++ وجود ندارند. برای مثال، می‌توانیم مستقیماً به مقایسه بردارها پرداخته و یک بردار را به بردار دیگری تخصیص دهیم. در فصل ۱۱ با نحوه پیاده سازی آرایه‌ها بصورت کلاس‌های تعریف شده توسط کاربر آشنا خواهید شد. این تعریف جدید از آرایه امکان می‌دهد تا کل آرایه را با دستوارت cin و cout وارد و خارج کرده و آرایه‌ها را به هنگام ایجاد مقداردهی اولیه کرده از دسترسی به خارج از محدودهٔ عناصر آرایه اجتناب کرده و

شاخص‌ها را تغییر داد (حتی نوع شاخص‌ها). از اینرو نیازی نیست که اولین عنصر آرایه، عنصر صفر باشد. حتی می‌توانیم از شاخص‌های غیرصحیح استفاده کنیم.

اجتناب از خطا

در فصل ۱۱، با نحوه ایجاد کلاس‌های که نشاندهنده آرایه‌های هوشمند هستند آشنا خواهید شد، که در زمان اجرا مبادرت به بررسی مرزهای آرایه می‌کنند. با استفاده از چنین آرایه‌های می‌توان جلوی برخی از خطاها را گرفت.

استفاده از آرایه‌های کاراکتری برای ذخیره‌سازی و کنترل رشته‌ها

تا بدین مرحله، فقط در مورد آرایه‌های صحیح صحبت کرده‌ایم. با این همه، امکان دارد آرایه‌های از نوع‌های مختلف داشته باشیم. در این بخش به معرفی نحوه ذخیره‌سازی رشته‌های کاراکتری در آرایه‌های کاراکتری می‌پردازیم. بخاطر دارید که، در ابتدای فصل سوم، از شی‌های **string** برای ذخیره سازی رشته‌های کاراکتری همانند نام دوره در کلاس **GradeBook** استفاده کردیم. رشته‌ای همانند "hello" در واقع یک آرایه کاراکتری است. در حالیکه شی‌های **string** برای کاهش خطا مناسب هستند، آرایه‌های کاراکتری که نشاندهنده رشته‌ها می‌باشند دارای ویژگیهای منحصر بفردی هستند که در این بخش با آنها آشنا خواهید شد. همانطوری که به یادگیری C++ ادامه می‌دهید، با قابلیت‌های C++ مواجه خواهید شد که استفاده از آرایه‌های کاراکتری را لازم می‌کنند. همچنین امکان دارد کدهای موجود را برای استفاده از آرایه‌های کاراکتری به روز کنید.

یک آرایه کاراکتری را می‌توان با استفاده از یک رشته لیترال مقداردهی اولیه کرد. برای مثال، اعلان

```
char string1[] = "first";
```

عناصر آرایه **string1** را با کاراکترهای جداگانه در رشته لیترال "first" مقداردهی می‌کند. سایز آرایه **string1** در اعلان فوق توسط کامپایلر و برپایه طول رشته تعیین می‌شود. توجه به این نکته مهم است که رشته "first" حاوی پنج کاراکتر به همراه یک کاراکتر پایان دهنده رشته بنام کاراکتر null است. بنابر این آرایه **string1** حاوی شش عنصر می‌باشد. ثابت کاراکتری نشاندهنده کاراکتر **null** رشته '**\0**' است (یک خط مورب و بدنبال آن صفر). تمام رشته‌های عرضه شده توسط آرایه‌های کاراکتری با این کاراکتر خاتمه می‌یابند. یک آرایه کاراکتری که عرضه‌کننده رشته است بایستی به میزان کافی بزرگ اعلان شده باشد تا بتواند کاراکترهای موجود در رشته را به همراه کاراکتر **null** نگهداری کند.

همچنین آرایه‌های کاراکتری را می‌توان با ثابت‌های کاراکتری مجزا از هم در یک لیست مقداردهی اولیه، مقداردهی کرد. اعلان زیر معادل با اعلان فوق است.

```
char string1[] = {'f' , 'i' , 'r' , 's' , 't' , '\0'};
```

به گوتیشن‌های قرار گرفته در اطراف هر ثابت کاراکتری توجه کنید. همچنین توجه کنید که بصورت صریح کاراکتر **null** در لیست مقداردهی اولیه تدارک دیده شده است. بدون آن، این آرایه فقط نشاندهنده، آرایه‌ای از کاراکترهاست، نه یک رشته. همانطوری که در فصل هشتم شاهد خواهید بود، تدارک ندیدن کاراکتر **null** برای یک رشته می‌تواند خطای منطقی بوجود آورد.

بدلیل اینکه یک رشته یک آرایه کاراکتری است، می‌توانیم به کاراکترهای مجزا یک رشته و به کمک شاخص آرایه مستقیماً دسترسی پیدا کنیم. برای مثال، **string1[0]** کاراکتر **'f'**، **string1[3]** کاراکتر **'s'** و **string1[5]** کاراکتر **null** است.

همچنین می‌توانیم یک رشته را مستقیماً به یک آرایه کاراکتری از طریق صفحه‌کلید و با استفاده از دستور **cin** و << وارد کنیم. برای مثال، اعلان

```
char string2[20];
```

یک آرایه کاراکتری با ظرفیت نگهداری 19 کاراکتر و یک کاراکتر **null** ایجاد می‌کند، عبارت

```
cin >> string2;
```

رشته‌ای از صفحه کلید بدرون **string2** خوانده و کاراکتر **null** را به انتهای رشته وارد شده توسط کاربر الصاق می‌کند. توجه کنید که در عبارت فوق فقط نام آرایه بدون هیچ‌گونه اطلاعاتی در مورد سایز آرایه بکار گرفته شده است. این وظیفه برنامه‌نویس است تا مطمئن شود که آرایه مورد نظر قادر به نگهداری رشته تایپ شده از سوی کاربر است. بطور پیش‌فرض، **cin** کاراکترها را از صفحه‌کلید تا رسیدن به اولین کاراکتر *white-space* می‌خواند (صرف نظر از سایز آرایه). از اینرو، وارد کردن داده با **cin** و << می‌تواند داده را بیش از مرز فوقانی آرایه وارد سازد (بخش ۱۳–۸ در این ارتباط است).

خطای برنامه‌نویسی

*تدارک ندیدن <<**cin** به همراه یک آرایه که بمیزان کافی برای ذخیره سازی رشته تایپ شده توسط صفحه کلید بزرگ نباشد، می‌تواند سبب از دست رفتن داده‌ها در برنامه شده و خطاهای جدی در زمان اجرا بوجود آورد.*

یک آرایه کاراکتری با یک رشته خاتمه یافته با null می‌تواند با دستور **cout** و >> چاپ شود. عبارت

```
cout << string2;
```

آرایه string2 را چاپ می‌کند. توجه کنید که >> **cout** همانند << **cin** توجهی به میزان بزرگی آرایه کاراکتری ندارد. کاراکترهای رشته تا رسیدن به کاراکتر null به خروجی ارسال می‌شوند.

برنامه شکل ۱۲–۷ به توصیف نحوه مقداردهی اولیه یک آرایه کاراکتری با یک رشته لیترال، خواندن یک رشته به یک آرایه کاراکتری، چاپ آرایه کاراکتری بصورت یک رشته و دسترسی به کاراکترهای جداگانه رشته پرداخته است.

```
1  // Fig. 7.12: fig07_12.cpp
2  // Treating character arrays as strings.
3  #include <iostream>
```

```
4  using std::cout;
5  using std::cin;
6  using std::endl;
7
8  int main()
9  {
10    char string1[ 20 ]; // reserves 20 characters
11    char string2[] = "string literal"; // reserves 15 characters
12
13    // read string from user into array string1
14    cout << "Enter the string \"hello there\": ";
15    cin >> string1; // reads "hello" [space terminates input]
16
17    // output strings
18    cout << "string1 is: " << string1 << "\nstring2 is: " << string2;
19
20    cout << "\nstring1 with spaces between characters is:\n";
21
22    // output characters until null character is reached
23    for ( int i = 0; string1[ i ] != '\0'; i++ )
24       cout << string1[ i ] << ' ';
25
26    cin >> string1; // reads "there"
27    cout << "\nstring1 is: " << string1 << endl;
28
29    return 0; // indicates successful termination
30 } // end main
```

```
Enter the string "hello there" : hello there
string1 is: hello
string2 is: string literal
String1 with spaces between character is:
h e l l o
string1 is: there
```

شکل ۱۲-۷ آرایه‌های کاراکتری برای پردازش رشته‌ها.

در خطوط 24-23 از شکل ۱۲-۷ از یک عبارت **for** برای ایجاد حلقه در میان آرایه **string1** و چاپ کاراکترهای مجزا شده توسط فاصله‌ها استفاده شده است. شرط موجود در عبارت **for**، شرط **string1[i] '\0'=!** تا مواجه شدن با کاراکتر **null** در رشته برقرار خواهد بود.

آرایه‌های محلی استاتیک و اتوماتیک

در فصل ششم در ارتباط با کلاس ذخیره‌سازی**static** بحث کردیم. یک متغیر محلی استاتیک در تعریف یک تابع، در مدت زمان اجرای برنامه وجود خواهد داشت و در بدنه همان تابع قابل استفاده و رویت است.

کارائی

*می‌توانیم از **static** در اعلان یک آرایه محلی استفاده کنیم. در اینصورت دیگر آرایه در هر بار فراخوانی تابع توسط برنامه، ایجاد و مقداردهی نشده و در هر بار با خاتمه یافتن تابع، آرایه از بین نمی‌رود. اینکار می‌تواند سبب افزایش کارایی شود، بویژه به هنگام کار با آرایه‌های بزرگ.*

برنامه‌ها، آرایه محلی استاتیک را به هنگام اعلان آنها و اولین بار که با آنها برخورد می‌کنند، مقداردهی اولیه می‌نمایند. اگر یک آرایه استاتیک بصورت صریح توسط برنامه‌نویس مقداردهی اولیه نشود، هر

عنصر آن آرایه با صفر و توسط کامپایلر در زمان ایجاد آرایه مقداردهی خواهد شد. بخاطر داشته باشید C++ چنین مقداردهی اولیه‌ای را برای متغیرهای اتوماتیک انجام نمی‌دهد.

برنامه شکل۱۳-۷ به توصیف تابع **staticArrayInit** (خطوط 41-25) با یک آرایه استاتیک محلی (خط 28) و تابع **automaticArrayInit** (خطوط 60-44) با یک آرایه محلی (خط 47) پرداخته است.

```
1  // Fig. 7.13: fig07_13.cpp
2  // Static arrays are initialized to zero.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  void staticArrayInit( void ); // function prototype
8  void automaticArrayInit( void ); // function prototype
9
10 int main()
11 {
12    cout << "First call to each function:\n";
13    staticArrayInit();
14    automaticArrayInit();
15
16    cout << "\n\nSecond call to each function:\n";
17    staticArrayInit();
18    automaticArrayInit();
19    cout << endl;
20
21    return 0; // indicates successful termination
22 } // end main
23
24 // function to demonstrate a static local array
25 void staticArrayInit( void )
26 {
27    // initializes elements to 0 first time function is called
28    static int array1[ 3 ]; // static local array
29
30    cout << "\nValues on entering staticArrayInit:\n";
31
32    // output contents of array1
33    for ( int i = 0; i < 3; i++ )
34       cout << "array1[" << i << "] = " << array1[ i ] << "  ";
35
36    cout << "\nValues on exiting staticArrayInit:\n";
37
38    // modify and output contents of array1
39    for ( int j = 0; j < 3; j++ )
40       cout << "array1[" << j << "] = " << ( array1[ j ] += 5 ) << "  ";
41 } // end function staticArrayInit
42
43 // function to demonstrate an automatic local array
44 void automaticArrayInit( void )
45 {
46    // initializes elements each time function is called
47    int array2[ 3 ] = { 1, 2, 3 }; // automatic local array
48
49    cout << "\n\nValues on entering automaticArrayInit:\n";
50
51    // output contents of array2
52    for ( int i = 0; i < 3; i++ )
53       cout << "array2[" << i << "] = " << array2[ i ] << "  ";
54
55    cout << "\nValues on exiting automaticArrayInit:\n";
56
57    // modify and output contents of array2
58    for ( int j = 0; j < 3; j++ )
59       cout << "array2[" << j << "] = " << ( array2[ j ] += 5 ) << "  ";
60 } // end function automaticArrayInit
```

```
First call to each function:
Values on entering staticArrayInit:
arry1[0] = 0 array1[1] = 0 array1[2] = 0
Values on exiting staticarrayInit:
array1[0] = 5 array1[1] = 5 array1[2] = 5

Values on entering automaticArrayInit:
arry2[0] = 1 array2[1] = 2 array2[2] = 3
Values on exiting automaticArrayInit:
array2[0] = 6 array2[1] = 7 array2[2] = 8

Second call to each function:

Values on entering staticArrayInit:
arry1[0] = 5 array1[1] = 5 array1[2] = 5
Values on exiting staticArrayInit:
array1[0] = 10 array1[1] = 10 array1[2] = 10

Values on entering automaticArrayInit:
arry2[0] = 1 array2[1] = 2 array2[2] = 3
Values on exiting automaticArrayInit:
array2[0] = 6 array2[1] = 7 array2[2] = 8
```

شکل ۱۳-۷ | مقداردهی اولیه آرایه‌های استاتیک و اتوماتیک.

تابع **staticArrayInit** دو بار فراخوانی شده است (خطوط13و17). آرایه محلی استاتیک توسط کامپایلر در اولین فراخوانی تابع با صفر مقداردهی شده است. تابع، آرایه را چاپ کرده و به عنصر مقدار 5 را افزوده و مجدداً آرایه را چاپ می‌کند. بار دوم تابع فراخوانی می‌شود، آرایه استاتیک حاوی مقادیر تغییر یافته و ذخیره شده از اولین فراخوانی تابع است. تابع **automaticArrayInit** هم دو بار فراخوانی شده است (خطوط 19 و18). عناصر آرایه محلی اتوماتیک با مقادیر 1، 2 و 3 مقداردهی اولیه می‌شوند (خط 47). تابع آرایه را چاپ کرده، مقدار 5 به هر عنصر افزوده و آرایه را مجدداً چاپ می‌کند. بار دوم تابع فراخوانی می‌شود و عناصر آرایه مجدداً با 1، 2 و 3 مقداردهی اولیه می‌شوند. آرایه دارای کلاس ذخیره سازی اتوماتیک است، از اینرو در هر بار فراخوانی **automaticArrayInit** آرایه مجدداً ایجاد می‌شود.

خطای برنامه‌نویسی

فرض اینکه عناصر یک تابع با آرایه استاتیک محلی در هر بار فراخوانی تابع مقداردهی اولیه خواهند شد، می‌تواند شما را به سمت خطاهای منطقی در برنامه هدایت کند.

۷-۵ ارسال آرایه به توابع

برای ارسال آرایه بعنوان یک آرگومان به یک تابع، باید نام آرایه بدون استفاده از براکت‌ها مشخص شود. برای مثال، اگر آرایه **hourlyTemperatures** بصورت زیر اعلان شده باشد:

```
int hourlyTemperatures[ 24 ];
```

در فراخوانی تابع

```
modifyArray( hourlyTemperatures );
```

آرایه **hourlyTemperatures** به تابع **modifyArray** ارسال می‌شود. زمانیکه آرایه‌ای به یک تابع ارسال می‌شود، سایز آرایه هم به همراه آن ارسال می‌شود، از اینرو تابع قادر به پردازش تعداد مشخص از عناصر در آرایه خواهد بود. در بخش۱۱-۷ به هنگام معرفی کلاس **vector** شاهد خواهید بود که سایز یک vector حالت توکار داشته و هر شی از سایز خود مطلع است. بنابر این به هنگام ارسال یک شی vector به یک تابع، نیازی نیست تا سایز vector بعنوان یک آرگومان ارسال شود. ارسال آرایه‌ها به توابع در C++ بصورت مراجعه صورت می‌گیرد. تابع فراخوانی شده قادر به تغییر مقدار عناصر در آرایه اصلی خواهد بود.

مقدار نام آرایه آدرس حافظه‌ای در کامپیوتر است که نشاندهنده اولین عنصر آرایه می‌باشد. بدلیل اینکه آدرس شروع آرایه ارسال می‌شود، تابع فراخوانی شده بطور دقیق از مکان ذخیره شده آرایه در حافظه مطلع است. بنابر این، زمانیکه تابع فراخوانی شده مبادرت به تغییر عناصر آرایه در درون بدنه خود می‌کند، این تغییرات بر روی عناصر واقعی آرایه در مکان‌های اصلی آن در حافظه اعمال می‌شوند.

اگر چه کل آرایه بصورت مراجعه ارسال می‌شود، اما می‌توان عناصر جداگانه آرایه را به همان روش ساده‌ای که متغیرها ارسال می‌شوند، ارسال کرد. در آرایه‌ای که دارای عناصری از نوع داده اصلی همانند **int** است، می‌توان از روش ارسال با مقدار استفاده کرد و این امر بستگی به تعریف تابع دارد. به چنین واحدهای منفرد داده‌ای گاها موجودیت‌های *scalar* می‌گویند. برای ارسال یک عنصر آرایه به یک تابع، از نام شاخص عنصر آرایه بعنوان یک آرگومان در فراخوانی تابع استفاده می‌شود.

در تابعی که یک آرایه را از طریق فراخوانی تابع دریافت می‌کند، باید لیست پارامتری آن برای دریافت آرایه آماده شده باشد. برای مثال، سرآیند تابع **modifyArray** می‌تواند بصورت زیر نوشته شده باشد

```
void modifyArray( int[] b, int arraySize )
```

این اعلان نشان می‌دهد که **modifyArray** در انتظار دریافت یک آرایه از نوع صحیح برای پارامتر **b** و تعداد عناصر آرایه در پارامتر *arraySize* است. نیازی به قرار دادن سایز آرایه در میان براکت‌ها نیست. اگر چنین کاری انجام دهید توسط کامپایلر نادیده گرفته خواهد شد. چراکه C++ آرایه‌ها را به روش مراجعه به توابع ارسال می‌کند. زمانیکه تابع فراخوانی شده از آرایه پارامتری بنام **b** استفاده می‌کند، در واقع به سایز واقعی آن در فراخوان مراجعه می‌نماید.

به ظاهر عجیب نمونه اولیه تابع **modifyArray** توجه کنید

```
void modifyArray(int [], int);
```

این نمونه اولیه را می‌توان بصورت زیر هم نوشت

```
void modifyArray (int anyArrayName[], int anyVariableName);
```

اما همانطوری که در فصل سوم آموختیم، کامپایلرهای C++ اسامی متغیرها در نمونه‌های اولیه (*prototypes*) را درنظر نمی‌گیرند. بخاطر دارید که، نمونه اولیه به کامپایلر، تعداد آرگومان‌ها و نوع هر آرگومان را اعلان می‌کند (البته ترتیب آنها را هم نشان می‌دهد).

برنامه شکل ۱۴-۷ به توصیف تفاوت موجود مابین ارسال کل آرایه و عنصری از یک آرایه پرداخته است. خطوط 23-22 پنج عنصر اصلی از آرایه صحیح بنام **a** را چاپ می‌کنند. خط 28 آرایه **a** و سایز آنرا به تابع **modifyArray** ارسال می‌کند (خطوط 50-45) که هر عنصر **a** را در 2 ضرب می‌کند (از طریق پارامتر **b**). سپس، خطوط 33-32 آرایه **a** را مجدداً در **main** چاپ می‌کنند. همانطوری که در خروجی برنامه دیده می‌شود، عناصر a براستی توسط **modifyArray** تغییر یافته‌اند. سپس، خط 36 مقدار عددی a[3] را چاپ کرده، سپس خط 38 عنصر a[3] را به تابع **modifyElement** ارسال می‌کند (خطوط 54-58)، که آن پارامتر در 2 ضرب شده و مقدار جدید چاپ می‌شود. توجه کنید که به هنگام چاپ a[3] توسط خط 39 در **main**، مقدار تغییر نیافته است، چرا که عناصر جداگانه آرایه به روش مقدار ارسال می‌شوند.

```
1  // Fig. 7.14: fig07_14.cpp
2  // Passing arrays and individual array elements to functions.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <iomanip>
8  using std::setw;
9
10 void modifyArray( int [], int ); // appears strange
11 void modifyElement( int );
12
13 int main()
14 {
15    const int arraySize = 5; // size of array a
16    int a[ arraySize ] = { 0, 1, 2, 3, 4 }; // initialize array a
17
18    cout << "Effects of passing entire array by reference:"
19       << "\n\nThe values of the original array are:\n";
20
21    // output original array elements
22    for ( int i = 0; i < arraySize; i++ )
23       cout << setw( 3 ) << a[ i ];
24
25    cout << endl;
26
27    // pass array a to modifyArray by reference
28    modifyArray( a, arraySize );
29    cout << "The values of the modified array are:\n";
30
31    // output modified array elements
32    for ( int j = 0; j < arraySize; j++ )
33       cout << setw( 3 ) << a[ j ];
34
35    cout << "\n\n\nEffects of passing array element by value:"
36       << "\n\na[3] before modifyElement: " << a[ 3 ] << endl;
37
```

```
38    modifyElement( a[ 3 ] ); // pass array element a[ 3 ] by value
39    cout << "a[3] after modifyElement: " << a[ 3 ] << endl;
40
41    return 0; // indicates successful termination
42 } // end main
43
44 // in function modifyArray, "b" points to the original array "a" in memory
45 void modifyArray( int b[], int sizeOfArray )
46 {
47    // multiply each array element by 2
48    for ( int k = 0; k < sizeOfArray; k++ )
49       b[ k ] *= 2;
50 } // end function modifyArray
51
52 // in function modifyElement, "e" is a local copy of
53 // array element a[ 3 ] passed from main
54 void modifyElement( int e )
55 {
56    // multiply parameter by 2
57    cout << "Value of element in modifyElement: " << ( e *= 2 ) << endl;
58 } // end function modifyElement
```

```
Effects of passing entire array by reference:
The values of the original array are:
  0 1 2 3 4
The values of the modified array are:
  0 2 4 6 8

Effects of passing array element by value:

a[3] before modifyElement: 6
value of element in modifyElement: 12
a[3] after modifyElement: 6
```

شکل ۱۴-۷ | ارسال کل آرایه و عناصر جداگانه آرایه به توابع.

احتمال دارد شرایطی در برنامه پیش آید که نبایستی تابعی اجازه تغییر در عناصر آرایه را داشته باشد. نوع **const** توسط C++ تدارک دیده شده که می‌تواند برای جلوگیری از تغییر مقادیر آرایه در کد تابع فراخوانی شده، بکار گرفته شود. زمانیکه تابعی یک آرایه را به همراه **const** مشخص می‌کند، عناصر آرایه در بدنه تابع تبدیل به ثابت شده و هرگونه اقدام به تغییر عناصر آرایه در بدنه تابع، خطای کامپایل بدنبال خواهد داشت. اینکار به برنامه‌نویسان کمک می‌کند تا جلوی تغییرات ناخواسته در عناصر آرایه را در بدنه توابع بگیرند.

برنامه شکل ۱۵-۷ به بررسی عملکرد توصیف کننده **const** پرداخته است. تابع **tryToModifyArray** در خطوط 26-21 به همراه پارامتر **const int b[]** تعریف شده است که نشان می‌دهد آرایه b ثابت بوده و نمی‌تواند تغییر داده شود. نتیجه هر سه بار اقدام تابع به تغییر عناصر آرایه b (خطوط 25-23) خطای کامپایل است. برای مثال کامپایلر Microsoft Visual C++.NET خطای "l-value specifies const object." را تولید می‌کند. این پیغام خطا براین نکته دلالت دارد که استفاده از یک شی const (برای مثال، **b[0]**) بعنوان یک *lvalue* (مقدار سمت چپ) خطا است. نمی‌توانید یک مقدار جدید به یک شی **const** با قرار دادن آن در سمت چپ عملگر تخصیص، تخصیص دهید. توجه کنید که پیغام‌های خطا در میان کامپایلرها متفاوت

از هم است (همانند پیغام‌های به نمایش درآمده در برنامه شکل ۱۵-۷). در فصل ۱۰ مجدداً در ارتباط با توصیف کننده **const** صحبت خواهیم کرد.

خطای برنامه‌نویسی

نتیجه فراموش کردن این نکته که آرایه‌ها به صورت مراجعه ارسال می‌شوند و می‌توانند توسط توابع فراخوانی شده تغییر داده شوند، خطای منطقی بدنبال خواهد داشت.

مهندسی نرم‌افزار

*اعمال نوع **const** در کنار یک پارامتر آرایه در تعریف یک تابع برای اجتناب از تغییر عناصر آرایه در بدنه تابع فراخوانی شده، نمونه دیگری از قاعده واگذاری حداقل امتیاز است. توابع نبایستی قادر به تغییر دادن آرایه باشند، مگر اینکه به اینکار واقعاً نیاز باشد.*

```
1  // Fig. 7.15: fig07_15.cpp
2  // Demonstrating the const type qualifier.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  void tryToModifyArray( const int [] ); // function prototype
8
9  int main()
10 {
11    int a[] = { 10, 20, 30 };
12
13    tryToModifyArray( a );
14    cout << a[ 0 ] << ' ' << a[ 1 ] << ' ' << a[ 2 ] << '\n';
15
16    return 0; // indicates successful termination
17 } // end main
18
19 // In function tryToModifyArray, "b" cannot be used
20 // to modify the original array "a" in main.
21 void tryToModifyArray( const int b[] )
22 {
23    b[ 0 ] /= 2; // error
24    b[ 1 ] /= 2; // error
25    b[ 2 ] /= 2; // error
26 } // end function tryToModifyArray
```

Borland C++ command-line compiler error message:

```
Error E2024 fig07_15.ccp 23: Cannot modify a const object
   in function tryToModifyArray(const int * const)
Error E2024 fig07_15.ccp 24: Cannot modify a const object
   in function tryToModifyArray(const int * const)
Error E2024 fig07_15.ccp 25: Cannot modify a const object
   in function tryToModifyArray(const int * const)
```

Microsoft Visual C++ .NET compiler error message:

```
C:\cpphtp5_examples\ch07\fig07_15.ccp(23) : error C2166: l-value specifies
    const object
C:\cpphtp5_examples\ch07\fig07_15.ccp(24) : error C2166: l-value specifies
    const object
C:\cpphtp5_examples\ch07\fig07_15.ccp(25) : error C2166: l-value specifies
   const object
```

GNU C++ compiler error message:

```
fig07_15.ccp:23: error: assignment of read-only location
fig07_15.ccp:24: error: assignment of read-only location
fig07_15.ccp:25: error: assignment of read-only location
```

شکل ۱۵-۷ | اعمال نوع const بر روی یک پارامتر آرایه.

٦-۷ مبحث آموزشی: کلاس GradeBook با استفاده از آرایه برای ذخیره‌سازی نمرات

این بخش نسخه بهبود یافته‌ای از کلاس **GradeBook** را که در فصل سوم معرفی شد و در طول فصل‌های چهارم الی ششم توسعه یافته، عرضه می‌کند. بخاطر دارید که این کلاس یک دفترچه نمره را نشان می‌دهد که نمرات را ذخیره کرده و به تحلیل آنها می‌پردازد. نسخه‌های قبلی این کلاس مبادرت به پردازش مجموعه‌ای از نمرات وارد شده توسط کاربر می‌کردند، اما قادر به نگهداری مقادیر نمرات در اعضای داده کلاس نبودند. بنابر این، تکرار محاسبات نیازمند، ورود مجدد همان نمرات توسط کاربر بود. یک راه حل برای این مشکل، ذخیره هر نمره وارد شده در یک عضو داده مجزا از کلاس است. برای مثال، می‌توانیم اعضای داده grade1، grade2 ، ... ،grade10 را در کلاس **GradeBook** برای ذخیره نمره دانشجو ایجاد کنیم. با این همه، کدی که مجموع نمرات را محاسبه و میانگین کلاس را بدست می‌آورد می‌تواند بسیار درهم شود. در این بخش، این مشکل را با ذخیره‌سازی نمرات در یک آرایه حل می‌کنیم.

ذخیره‌سازی نمرات در آرایه‌ای از کلاس GradeBook

نسخه کلاس **GradeBook** عرضه شده در این بخش (برنامه‌های ۱۶-۷ و ۱۷-۷) از یک آرایه صحیح برای ذخیره‌سازی نمرات چندین دانشجو در یک امتحان، استفاده می‌کند. آرایه **grades** بصورت یک عضو داده در خط 29 از برنامه شکل ۱۶-۷ اعلان شده است، بنابر این هر شی **GradeBook** مجموعه نمرات متعلق بخود را نگهداری خواهدکرد.

```
1  // Fig. 7.16: GradeBook.h
2  // Definition of class GradeBook that uses an array to store test grades.
3  // Member functions are defined in GradeBook.cpp
4
5  #include <string> // program uses C++ Standard Library string class
6  using std::string;
7
8  // GradeBook class definition
9  class GradeBook
10 {
11 public:
12    // constant -- number of students who took the test
13    const static int students = 10; // note public data
14
15    // constructor initializes course name and array of grades
16    GradeBook( string, const int [] );
17
18    void setCourseName( string ); // function to set the course name
19    string getCourseName(); // function to retrieve the course name
20    void displayMessage(); // display a welcome message
21    void processGrades(); // perform various operations on the grade data
22    int getMinimum(); // find the minimum grade for the test
23    int getMaximum(); // find the maximum grade for the test
24    double getAverage(); // determine the average grade for the test
25    void outputBarChart(); // output bar chart of grade distribution
26    void outputGrades(); // output the contents of the grades array
27 private:
28    string courseName; // course name for this grade book
29    int grades[ students ]; // array of student grades
```

```
30 }; // end class GradeBook
```

شکل ۱۶-۷ | تعریف کلاس GradeBook با استفاده از آرایه برای ذخیره نمرات.

```
1   // Fig. 7.17: GradeBook.cpp
2   // Member-function definitions for class GradeBook that
3   // uses an array to store test grades.
4   #include <iostream>
5   using std::cout;
6   using std::cin;
7   using std::endl;
8   using std::fixed;
9
10  #include <iomanip>
11  using std::setprecision;
12  using std::setw;
13
14  #include "GradeBook.h" // GradeBook class definition
15
16  // constructor initializes courseName and grades array
17  GradeBook::GradeBook( string name, const int gradesArray[] )
18  {
19     setCourseName( name ); // initialize courseName
20
21     // copy grades from gradeArray to grades data member
22     for ( int grade = 0; grade < students; grade++ )
23        grades[ grade ] = gradesArray[ grade ];
24  } // end GradeBook constructor
25
26  // function to set the course name
27  void GradeBook::setCourseName( string name )
28  {
29     courseName = name; // store the course name
30  } // end function setCourseName
31
32  // function to retrieve the course name
33  string GradeBook::getCourseName()
34  {
35     return courseName;
36  } // end function getCourseName
37
38  // display a welcome message to the GradeBook user
39  void GradeBook::displayMessage()
40  {
41     // this statement calls getCourseName to get the
42     // name of the course this GradeBook represents
43     cout << "Welcome to the grade book for\n" << getCourseName() << "!"
44        << endl;
45  } // end function displayMessage
46
47  // perform various operations on the data
48  void GradeBook::processGrades()
49  {
50     // output grades array
51     outputGrades();
52
53     // call function getAverage to calculate the average grade
54     cout << "\nClass average is " << setprecision( 2 ) << fixed <<
55        getAverage() << endl;
56
57     // call functions getMinimum and getMaximum
58     cout << "Lowest grade is " << getMinimum() << "\nHighest grade is "
59        << getMaximum() << endl;
60
61     // call function outputBarChart to print grade distribution chart
62     outputBarChart();
63  } // end function processGrades
64
65  // find minimum grade
66  int GradeBook::getMinimum()
```

```
67  {
68     int lowGrade = 100; // assume lowest grade is 100
69
70     // loop through grades array
71     for ( int grade = 0; grade < students; grade++ )
72     {
73        // if current grade lower than lowGrade, assign it to lowGrade
74        if ( grades[ grade ] < lowGrade )
75           lowGrade = grades[ grade ]; // new lowest grade
76     } // end for
77
78     return lowGrade; // return lowest grade
79  } // end function getMinimum
80
81  // find maximum grade
82  int GradeBook::getMaximum()
83  {
84     int highGrade = 0; // assume highest grade is 0
85
86     // loop through grades array
87     for ( int grade = 0; grade < students; grade++ )
88     {
89        // if current grade higher than highGrade, assign it to highGrade
90        if ( grades[ grade ] > highGrade )
91           highGrade = grades[ grade ]; // new highest grade
92     } // end for
93
94     return highGrade; // return highest grade
95  } // end function getMaximum
96
97  // determine average grade for test
98  double GradeBook::getAverage()
99  {
100    int total = 0; // initialize total
101
102    // sum grades in array
103    for ( int grade = 0; grade < students; grade++ )
104       total += grades[ grade ];
105
106    // return average of grades
107    return static_cast< double >( total ) / students;
108 } // end function getAverage
109
110 // output bar chart displaying grade distribution
111 void GradeBook::outputBarChart()
112 {
113    cout << "\nGrade distribution:" << endl;
114
115    // stores frequency of grades in each range of 10 grades
116    const int frequencySize = 11;
117    int frequency[ frequencySize ] = { 0 };
118
119    // for each grade, increment the appropriate frequency
120    for ( int grade = 0; grade < students; grade++ )
121       frequency[ grades[ grade ] / 10 ]++;
122
123    // for each grade frequency, print bar in chart
124    for ( int count = 0; count < frequencySize; count++ )
125    {
126       // output bar labels ("0-9:", ..., "90-99:", "100:" )
127       if ( count == 0 )
128          cout << "  0-9: ";
129       else if ( count == 10 )
130          cout << "  100: ";
131       else
132          cout << count * 10 << "-" << ( count * 10 ) + 9 << ": ";
133
134       // print bar of asterisks
135       for ( int stars = 0; stars < frequency[ count ]; stars++ )
136          cout << '*';
```

```
137
138        cout << endl; // start a new line of output
139     } // end outer for
140  } // end function outputBarChart
141
142  // output the contents of the grades array
143  void GradeBook::outputGrades()
144  {
145    cout << "\nThe grades are:\n\n";
146
147     // output each student's grade
148     for ( int student = 0; student < students; student++ )
149        cout << "Student " << setw( 2 )<< student + 1 <<": " << setw( 3 )
150           << grades[ student ] << endl;
151  } // end function outputGrades
```

شکل ۱۷-۷ | توابع عضو برای کار با آرایه نمرات.

به سایز آرایه در خط 29 از شکل ۱۶-۷ توجه کنید که در آن عضو داده **students** بصورت *سراسری* (*public*) و **const static** مشخص شده است (اعلان شده در خط 13). این عضو داده سراسری است، از اینروست که از در دسترس سرویس‌گیرنده‌های کلاس قرار دارد. بزودی شاهد مثالی از یک برنامه سرویس‌گیرنده خواهید بود که از این ثابت استفاده می‌کند. اعلان **students** با توصیف کننده **const** نشان می‌دهد که این عضو داده ثابت است، مقدار آن پس از مقداردهی اولیه قابل تغییر نخواهد بود. کلمه کلیدی **static** در اعلان این متغیر بر این نکته دلالت دارد که عضو داده در میان تمام شی‌ها، کلاس به اشتراک گذاشته خواهد شد، تمام شی‌های **GradeBook** به تعداد دانشجویان نمره ذخیره خواهند کرد. از بخش ۶-۳ بخاطر دارید، زمانیکه هر شی از کلاس از یک صفت کپی شده خود نگهداری می‌کند، متغیری که نشاندهنده صفت است بعنوان یک عضو داده شناخته می‌شود، هر شی (نمونه) از کلاس دارای یک کپی مجزا از متغیر در حافظه است. آنها متغیرهای برای هر شی از کلاس هستند که دارای یک کپی مجزا نیستند. اینحالت با اعضای داده **static** رخ می‌دهد که بعنوان متغیرهای کلاس هم شناخته می‌شوند. زمانیکه شی‌های یک کلاس حاوی عضوهای داده **static** ایجاد می‌شوند، تمام شی‌های آن کلاس یک کپی از عضوهای داده **static** را به اشتراک می‌گذارند. یک عضو داده **static** می‌تواند از طریق تعریف کلاس و تعریف تابع عضو همانند هر عضو داده دیگر، در دسترس قرار گیرد. همانطوری که بزودی شاهد خواهید بود یک عضو داده سراسری **static** می‌تواند از خارج از کلاس در دسترس قرار گیرد، حتی زمانیکه هیچ شی از کلاس وجود نداشته باشد. اینکار با استفاده از نام کلاس و بدنبال آن عملگر تفکیک قلمرو باینری (**::**) و نام عضو داده صورت می‌گیرد. در فصل دهم با اعضای داده **static** بیشتر آشنا خواهید شد.

سازنده کلاس (اعلان شده در خط 16 از شکل ۱۶-۷ و تعریف شده در خطوط 24-17 از شکل ۱۷-۷) دارای دو پارامتر است، نام دوره و یک آرایه از نمرات. زمانیکه برنامه یک شی **GradeBook** ایجاد می‌کند (خط 13 از fig07_18.cpp)، برنامه یک آرایه از نوع **int** را به سازنده ارسال می‌کند، که مقادیر

موجود در آرایه ارسالی را به عضو **grades** کپی می‌کند (خطوط 23-22 از شکل ۱۷–۷). مقادیر نمرات در آرایه ارسالی می‌توانند از طریق کاربر یا از طریق یک فایل وارد برنامه شده باشند. در برنامه تست به نمایش درآمده، فقط یک آرایه با مجموعه‌ای از نمرات را مقداردهی کرده‌ایم (خطوط 11-10 از شکل ۱۸–۷). پس از ذخیره نمرات در عضو داده **grades** از کلاس **GradeBook**، تمام توابع عضو کلاس در صورت نیاز قادر به دسترسی به آرایه **grades** خواهند بود تا محاسبات مورد نظر را انجام دهند.

```
1  // Fig. 7.18: fig07_18.cpp
2  // Creates GradeBook object using an array of grades.
3
4  #include "GradeBook.h" // GradeBook class definition
5
6  // function main begins program execution
7  int main()
8  {
9     // array of student grades
10    int gradesArray[ GradeBook::students ] =
11       { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
12
13    GradeBook myGradeBook(
14       "CS101 Introduction to C++ Programming", gradesArray );
15    myGradeBook.displayMessage();
16    myGradeBook.processGrades();
17    return 0;
18 } // end main
```

```
Welcome to the grade book for
CS101 Introduction to C++ Programming!

The grades are:

Student   1: 87
Student   2: 68
Student   3: 94
Student   4: 100
Student   5: 83
Student   6: 78
Student   7: 85
Student   8: 91
Student   9: 76
Student  10: 87

Class average is 84.90
Lowest grade is 68
Highest grade is 100

Grade distribution:
  0-9:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: *
70-79: **
80-89: ****
90-99: **
  100: *
```

شکل ۱۸-۷ | ایجاد یک شی GradeBook با استفاده از آرایه نمرات، سپس فراخوانی تابع عضو processGrades برای تحلیل نمرات.

تابع عضو **processGrade** (اعلان شده در خط 21 از شکل ۱۶-۷ و تعریف شده در خطوط 63-48 از شکل ۱۷-۷) حاوی دنباله‌ای از فراخوانی تابع عضو است که گزارشی از تحلیل نمرات چاپ می‌کند. خط 51 تابع عضو **outputGrades** را برای چاپ محتویات آرایه **grades** فراخوانی کرده است. خطوط 148- 150 در تابع عضو **outputGrades** از یک عبارت **for** برای چاپ نمره هر دانشجو استفاده کرده است. اگر چه شاخص آرایه از صفر شروع می‌شود، اما مایل هستیم تعداد دانشجویان از 1 آغاز شود. بنابر این خطوط 150-149 مقدار **student+1** را بعنوان شماره دانشجو در تولید برچسب‌های **"Student 1:"** ، **"Student 2:"** ، و الی آخر چاپ می‌کنند.

سپس تابع عضو **processGrade** مبادرت به فراخوانی تابع عضو **getAverage** (خطوط 55-54) برای بدست آوردن میانگین نمرات در آرایه می‌کند. تابع عضو **getAverage** (اعلان شده در خط 24 از شکل ۱۶-۷ و تعریف شده در خطوط 108-98) از یک عبارت **for** برای بدست آوردن مجموع مقادیر موجود در آرایه **grades** قبل از انجام محاسبه میانگین استفاده کرده است. توجه کنید که محاسبه میانگین بکار رفته در خط 107 از **const static** برای عضو داده **students** به منظور تعیین تعداد نمرات استفاده کرده است.

خطوط 59-58 در تابع عضو **processGrades** توابع **getMinimum** و **getMaximum** را برای تعیین پایین‌ترین و بالاترین نمرات هر دانشجو در امتحان فراخوانی می‌کنند. اجازه دهید تا به بررسی نحوه عملکرد تابع عضو **getMinimum** در یافتن کمترین نمره بپردازیم. بدلیل اینکه بالاترین نمره می‌تواند 100 باشد، فرض می‌کنیم که 100 کمترین نمره است (خط 68). سپس مبادرت به مقایسه هر عنصر آرایه با کمترین نمره می‌کنیم، تا مقادیر کمتر پیدا شوند. خطوط 76-71 در تابع **getMinimum** در میان آرایه حرکت کرده (حلقه) و خطوط 75-74 هر نمره را با مقدار **lowGrade** مقایسه می‌کند. اگر نمره کمتر از **lowGrade** باشد، **lowGrade** با آن نمره مقداردهی می‌شود. زمانیکه خط 78 اجرا شود، متغیر **lowGrade** حاوی کمترین نمره در آرایه خواهد بود. عملکرد تابع عضو **getMaximum** (خطوط 82- 95) مشابه تابع عضو **getMinimum** است. سرانجام، خط 62 در تابع عضو **processGrades** ، تابع عضو **outputBarChart** را برای چاپ نمودار توزیعی از نمرات را با تکنیک مشابه بکار رفته در برنامه شکل ۹-۷ فراخوانی می‌کند. در آن مثال، بصورت غیراتوماتیک تعداد نمرات در هر رده را محاسبه می‌کردیم (یعنی 9-0 ، 19-10 ، ... ، 99-90). در این مثال، خطوط 121-120 از تکنیک مشابهی که در برنامه‌های ۱۰-۷ و ۱۱-۷ در محاسبه تکرار نمرات در هر رده بکار برده شده‌اند، استفاده می‌کنند. در خط 117 آرایه

frequency با 11 عنصر از نوع صحیح برای ذخیره سازی فراوانی نمرات در هر رده از نمرات، اعلان و ایجاد شده است. برای هر نمره در آرایه **grades**، خطوط 121-120 مقدار عنصر مقتضی در آرایه **frequency** را افزایش می‌دهند. برای تعیین اینکه کدام عنصر افزایش خواهد یافت، خط 121 مبادرت به تقسیم نمره (**grade**) جاری بر 10 می‌کند (با استفاده از تقسیم صحیح). برای مثال، اگر نمره برابر 85 باشد، خط 121 مقدار **frequency[8]** را برای به روز کردن تعداد نمرات در محدوده 89-80 افزایش می‌دهد. سپس خطوط 139-124 نمودار میله‌ای را برپایه مقادیر در آرایه **frequency** چاپ می‌کنند (به شکل ۱۸-۷ نگاه کنید). همانند خطوط 30-29 از برنامه شکل ۹-۷، خطوط 136-135 از برنامه شکل ۱۷-۷ از مقداری در آرایه **frequency** برای تعیین تعداد ستاره‌ها در نمایش هر میله استفاده می‌کنند.

تست کلاس GradeBook

برنامه شکل ۱۸-۷ یک شی از کلاس **GradeBook** (شکل‌های ۱۶-۷ و ۱۷-۷) را با استفاده از آرایه **gradesArray** ایجاد می‌کند (اعلان و مقداردهی شده در خطوط 11-10). توجه کنید که از عملگر تفکیک قلمرو باینری (::) در عبارت "`GradeBook::students`" برای دسترسی به ثابت استاتیک **students** از کلاس **GradeBook** استفاده کرده‌ایم (خط 10). از این ثابت در اینجا برای ایجاد آرایه‌ای استفاده کرده‌ایم که هم سایز آرایه **grades** ذخیره شده بعنوان یک عضو داده در کلاس **GradeBook** باشد. خطوط 14-13 نام دوره و **gradeArray** را به سازنده **GradeBook** ارسال می‌کنند. خط 15 پیغام خوش‌آمدگویی را چاپ کرده و خط 16 تابع عضو **processGrades** را فراخوانی می‌کند. خروجی برنامه نشاندهنده تحلیل 10 نمره در **myGradeBook** است.

۷-۷ جستجوی آرایه‌ها: جستجوی خطی

غالباً، برنامه‌ها با مقادیر عظیمی از اطلاعات ذخیره شده در آرایه‌ها کار می‌کنند. در چنین مواردی تعیین اینکه آیا آرایه‌ای حاوی مقداری برابر با مقدار کلید است، ضروری می‌باشد. فرآیند مشخص کردن مکان مقدار یک عنصر خاص در آرایه، جستجو نامیده می‌شود. در این بخش، به بررسی یک تکنیک جستجو، بنام جستجوی خطی خواهیم پرداخت. در تمرین ۳۳-۷ از شما خواسته شده تا نسخه بازگشتی روش جستجوی خطی را پیاده‌سازی کنید. در فصل ۲۰، به بررسی روش موثرتری بنام جستجوی باینری می‌پردازیم.

جستجوی خطی

تابع **linearSearch** در برنامه ۱۹-۷، برای انجام جستجوی خطی است. تابع **linearSearch** (خطوط 37-44) از یک ساختار **for** حاوی یک عبارت **if** برای مقایسه هر عنصر آرایه با کلید جستجو است (خط 40). اگر عناصر آرایه در حال جستجو، مرتب نباشند، تابع بطور متوسط کلید جستجو را با نیمی از عناصر آرایه مقایسه خواهد کرد. روش جستجوی خطی بر روی آرایه‌های کوچک یا آرایه‌های نامرتب بخوبی کار می‌کند. با این همه، در مورد آرایه‌های بزرگ، جستجوی خطی کارایی مناسبی ندارد. اگر آرایه مرتب شده باشد (عناصر آرایه ترتیب خاص دارند)، می‌توانید از تکنیک جستجوی باینری که در فصل ۲۰ با آن آشنا خواهید شد استفاده کنید.

```
1  // Fig. 7.19: fig07_19.cpp
2  // Linear search of an array.
3  #include <iostream>
4  using std::cout;
5  using std::cin;
6  using std::endl;
7
8  int linearSearch( const int [], int, int ); // prototype
9
10 int main()
11 {
12    const int arraySize = 100; // size of array a
13    int a[ arraySize ]; // create array a
14    int searchKey; // value to locate in array a
15
16    for ( int i = 0; i < arraySize; i++ )
17       a[ i ] = 2 * i; // create some data
18
19    cout << "Enter integer search key: ";
20    cin >> searchKey;
21
22    // attempt to locate searchKey in array a
23    int element = linearSearch( a, searchKey, arraySize );
24
25    // display results
26    if ( element != -1 )
27       cout << "Found value in element " << element << endl;
28    else
29       cout << "Value not found" << endl;
30
31    return 0; // indicates successful termination
32 } // end main
33
34 // compare key to every element of array until location is
35 // found or until end of array is reached; return subscript of
36 // element if key or -1 if key not found
37 int linearSearch( const int array[], int key, int sizeOfArray )
38 {
39    for ( int j = 0; j < sizeOfArray; j++ )
40       if ( array[ j ] == key ) // if found,
41          return j; // return location of key
42
43    return -1; // key not found
44 } // end function linearSearch
```

```
Enter integer search key: 36
Found value in element 18
```

```
Enter integer search key: 37
Value not found
```

شکل ۱۹-۷ | جستجوی خطی در آرایه.

۸-۷ مرتب‌سازی آرایه‌ها

مرتب‌سازی داده‌ها (ترتیب دهی داده‌ها به یک روش مشخص همانند مرتب‌سازی صعودی یا نزولی) از اعمالی است که در بیشتر برنامه‌ها صورت می‌گیرد. برای مثال، یک بانک اقدام به مرتب کردن تمام چک‌ها با شماره حساب می‌کند و از اینرو می‌تواند صورت حساب‌های بانکی جداگانه‌ای در پایان هر ماه مهیا نماید. شرکت‌های تلفن لیست صورت‌حساب‌های خود را با نام خانوادگی مرتب می‌کنند تا یافتن شماره‌های تلفن آسانتر شود. در واقع هر سازمانی نیاز دارد تا به مرتب‌سازی داده‌های خود اقدام کند. مرتب‌سازی یکی از مسائل پیچیده در علم کامپیوتر است که تحقیقات فراوانی در این زمینه صورت گرفته است. در این بخش به بررسی یکی از ساده‌ترین طرح‌های مرتب‌سازی می‌پردازیم. در تمرینات انتهای این فصل و فصل ۲۰، به توضیح یک الگوریتم مرتب‌سازی پیچیده خواهیم پرداخت.

کارایی

گاهی اوقات، الگوریتم‌های ساده از کارایی پایینی برخوردار هستند. خاصیت چنین الگوریتم‌های در نوشتن آسان، تست و خطایابی است. ممکن است الگوریتم‌های پیچیده برای استفاده در برنامه‌های که نیاز به حداکثر کارایی دارند، بکار گرفته شوند.

مرتب‌سازی درجی

برنامه شکل ۲۰-۷ مقادیر 10 عنصر آرایه **data** را به ترتیب صعودی مرتب می‌کند. از تکنیکی بنام *مرتب‌سازی درجی* استفاده می‌کنیم، این تکنیک ساده بوده اما از کارایی کافی برخوردار نیست. در اولین تکرار این الگوریتم، دومین عنصر برداشته شده و اگر کوچکتر از اولین عنصر باشد، جای آنرا با اولین عنصر عوض می‌کند (یعنی برنامه دومین عنصر را در قبل از اولین عنصر درج می‌کند). در تکرار دوم به مقایسه سومین عنصر پرداخته و آنرا در مکان یا موقعیت صحیح با توجه به دو عنصر اول قرار می‌دهد، از اینرو هر سه عنصر مرتب شده‌اند. در تکرار i^{th} این الگوریتم، اولین عناصر *i*ام در آرایه مرتب شده خواهند بود.

خط 13 از برنامه شکل ۲۰-۷ مبادرت به اعلان و مقداردهی اولیه آرایه data با مقادیر زیر می‌کند:

```
34  56  4  10  77  51  93  30  5  52
```

ابتدا برنامه به **data[0]** و **data[1]** نگاه می‌کند، که دارای مقادیر 34 و 56 هستند. در حال حاضر این دو عنصر مرتب می‌باشند، از اینرو برنامه بکار ادامه می‌دهد، اگر مقادیر مرتب نباشند، برنامه جای آنها را تعویض می‌کند.

```
1  // Fig. 7.20: fig07_20.cpp
2  // This program sorts an array's values into ascending order.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
```

```
7  #include <iomanip>
8  using std::setw;
9
10 int main()
11 {
12    const int arraySize = 10; // size of array a
13    int data[ arraySize ] = { 34, 56, 4, 10, 77, 51, 93, 30, 5, 52 };
14    int insert; // temporary variable to hold element to insert
15
16    cout << "Unsorted array:\n";
17
18    // output original array
19    for ( int i = 0; i < arraySize; i++ )
20       cout << setw( 4 ) << data[ i ];
21
22    // insertion sort
23    // loop over the elements of the array
24    for ( int next = 1; next < arraySize; next++ )
25    {
26       insert = data[ next ]; // store the value in the current element
27
28       int moveItem = next; // initialize location to place element
29
30     // search for the location in which to put the current element
31       while ( ( moveItem > 0 ) && ( data[ moveItem - 1 ] > insert ) )
32       {
33          // shift element one slot to the right
34          data[ moveItem ] = data[ moveItem - 1 ];
35          moveItem--;
36       } // end while
37
38       data[ moveItem ] = insert; // place inserted element into the array
39    } // end for
40
41    cout << "\nSorted array:\n";
42
43    // output sorted array
44    for ( int i = 0; i < arraySize; i++ )
45       cout << setw( 4 ) << data[ i ];
46
47    cout << endl;
48    return 0; // indicates successful termination
49 } // end main
```

```
Unsorted array:
  34 56 4 10 77 51 93 30 5 52
Sorted array:
  4 5 10 30 34 51 52 56 77 93
```

شکل ۲۰-۷ | مرتب سازی آرایه به روش درجی.

در تکرار دوم، برنامه به **data[2]** با مقدار 4 نگاه می‌کند. این مقدار کمتر از 56 است، از اینرو برنامه 4 را در متغیر موقتی ذخیره کرده و 56 را یک عنصر به سمت راست حرکت می‌دهد. سپس برنامه تعیین می‌کند که 4 کمتر از 34 است و بنابر این 34 را یک عنصر به راست حرکت می‌دهد. اکنون برنامه به ابتدای آرایه رسیده است، از اینرو 4 در **data[0]** قرار داده شده است. وضعیت آرایه در این مرحله بصورت زیر است:

```
4  34  56  10  77  51  93  30  5  52
```

در تکرار سوم، برنامه مقدار **data[3]** یعنی 10 را در متغیر موقت ذخیره می‌سازد. سپس برنامه 10 را با 56 مقایسه کرده و 56 را یک عنصر به راست حرکت می‌دهد، چرا که بزرگتر از 10 است. سپس برنامه مبادرت به مقایسه 10 با 34 کرده، 34 را یک عنصر به راست حرکت می‌دهد. زمانیکه برنامه به مقایسه 10

با 4 می‌پردازد، متوجه می‌شود که 10 بزرگتر از 4 است و 10 را در **data[1]** قرار می‌دهد. اکنون آرایه بفرم زیر است:

```
4  10  34  56  77  51  93  30  5  52
```

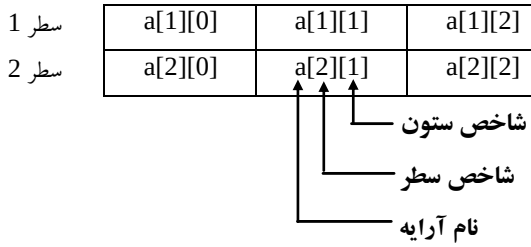
با استفاده از این الگوریتم ، در تکرار i^{th} ، اولین عنصر *i*ام از آرایه اصلی مرتب شده خواهند بود. عملیات مرتب سازی توسط عبارت **for** در خطوط 39-24 صورت می‌گیرد که در میان عناصر آرایه حرکت می‌کند. در هر تکرار خط 26 بصورت موقت مقدار عنصری که در بخش مرتب شده آرایه درج خواهد شد را در متغیر موقت **insert** ذخیره می‌کند (اعلان شده در خط 14). خط 28 مبادرت به اعلان و مقداردهی اولیه متغیر **moveItem** می‌کند که مسیر درج عنصر را نگهداری می‌کند. حلقه خطوط 36-31 مسئول یافتن موقعیت صحیح عنصری است که باید درج شود. حلقه زمانی خاتمه می‌پذیرد که برنامه به انتهای آرایه برسد یا به عنصری برسد که کمتر از مقدار درج شده است. خط 34 عنصر را به راست حرکت داده و خط 35 موقعیت را برای درج عنصر بعدی یک واحد کاهش می‌دهد. پس از اتمام حلقه **while**، خط 38 عنصر را در آن مکان درج می‌کند. زمانیکه عبارت **for** در خطوط 39-24 خاتمه یافت، عناصر آرایه مرتب شده خواهند بود.

اساس مرتب سازی درجی ساده بودن برنامه آن است، با این همه عملکرد کندی دارد. این سرعت کم در برخورد با آرایه‌های بزرگ بیشتر آشکار می‌شود. در تمرینات این فصل، تعدادی الگوریتم جایگزین برای مرتب سازی آرایه عرضه شده است. در فصل ۲۰ با تکنیک‌های کاربردی‌تر آشنا خواهید شد.

۹-۷ آرایه‌های چند بعدی

تا بدین جا با آرایه‌ای یک بعدی (یا یک شاخص‌دار) آشنا شده‌اید، که فقط حاوی یک سطر از مقادیر بودند. در این بخش، به توضیح آرایه‌های چند بعدی (یا آرایه‌های با چند شاخص) که نیاز به دو یا چند شاخص برای هویت دادن به عناصر آرایه دارند، خواهیم پرداخت. بحث ما بر روی آرایه‌های دوبعدی (یا دو شاخص‌دار) یا آرایه‌های که دارای چندین سطر از مقادیر هستند، متمرکز خواهد بود. طبق قرارداد برای مشخص کردن یکی از عناصر جدول، بایستی آنرا با دو شاخص نشان دهیم. شاخص اول، نشاندهنده سطر و شاخص دوم نشاندهنده ستون است. در شکل ۲۱-۷ یک آرایه دو بعدی منظم بنام **a**، حاوی سه سطر و چهار ستون دیده می‌شود. یک آرایه دو بعدی با m سطر و n ستون، یک آرایه n در m نامیده می‌شود.

	ستون 0	ستون 1	ستون 2	ستون 3
سطر 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]



شکل ۲۱-۷ | آرایه دو بعدی با سه سطر و چهار ستون.

هر عنصر در آرایه **a**، بصورت **a[i][j]** مشخص می‌شود، که در این فرم **a** نام آرایه، **j** و **i** شاخص‌های آن هستند که بصورت منحصر بفرد نشاندهنده سطر و ستون هر عنصر در آرایه **a** است. توجه کنید که بدلیل عملکرد شاخص‌ها بر مبنای شمارش از صفر، اسامی عناصر در سطر اول دارای شاخص، **0** هستند و اسامی عناصر در ستون چهارم دارای شاخص، **3** می‌باشند.

خطای برنامه‌نویسی

*مراجعه به یک عنصر آرایه دو بعدی **a[x][y]** بصورت **a[x , y]** خطا است. در واقع با **a[x , y]** بصورت **a[y]** رفتار می‌شود، چرا که **C++** عبارت **x,y** را فقط بعنوان **y** ارزیابی می‌کند.*

مقداردهی آرایه‌های چند بعدی در اعلان‌ها همانند آرایه‌های یک بعدی صورت می‌گیرد. برای مثال، یک آرایه دو بعدی منظم بنام **b** با مقادیر 1 و 2 در سطر 0 و مقادیر 3 و 4 در سطر 1 را می‌توان بصورت زیر اعلان و مقداردهی اولیه کرد:

```
int b[2][2] = {{1,2},{3,4}};
```

مقادیر گروه‌بندی شده با سطر در درون براکت‌ها، اقدام به مقداردهی **b[0][0]** با **1** و **b[0][1]** با **2** می‌کنند و **b[1][0]** با **3** و **b[1][1]** با **4** مقداردهی می‌شود. در آرایه‌های منظم، هر سطر دارای شمارهٔ یکسان است.

برنامه شکل ۲۲-۷، نحوه مقداردهی اولیه یک آرایه در زمان اعلان را نشان می‌دهد و همچنین با استفاده از حلقه‌های تودرتوی **for** اقدام به حرکت در میان عناصر آرایه‌ها شده است.

```
1  // Fig. 7.22: fig07_22.cpp
2  // Initializing multidimensional arrays.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  void printArray( const int [][ 3 ] ); // prototype
8
9  int main()
10 {
11    int array1[ 2 ][ 3 ] = { { 1, 2, 3 }, { 4, 5, 6 } };
12    int array2[ 2 ][ 3 ] = { 1, 2, 3, 4, 5 };
```

```
13    int array3[ 2 ][ 3 ] = { { 1, 2 }, { 4 } };
14
15    cout << "Values in array1 by row are:" << endl;
16    printArray( array1 );
17
18    cout << "\nValues in array2 by row are:" << endl;
19    printArray( array2 );
20
21    cout << "\nValues in array3 by row are:" << endl;
22    printArray( array3 );
23    return 0; // indicates successful termination
24 } // end main
25
26 // output array with two rows and three columns
27 void printArray( const int a[][ 3 ] )
28 {
29    // loop through array's rows
30    for ( int i = 0; i < 2; i++ )
31    {
32       // loop through columns of current row
33       for ( int j = 0; j < 3; j++ )
34          cout << a[ i ][ j ] << ' ';
35
36       cout << endl; // start new line of output
37    } // end outer for
38 } // end function printArray
```

```
Values in array1 by row are:
1 2 3
4 5 6

Values in array1 by row are:
1 2 3
4 5 0
Values in array1 by row are:
1 2 0
4 0 0
```

شکل ۲۲-۷ | مقداردهی آرایه‌های چند بعدی.

در اعلان آرایه **array1** (خط 11) اقدام به مقداردهی شش عنصر در دو زیر لیست شده است. اولین زیرلیست، سطر اول (سطر صفر) از آرایه را با مقادیر **3** و **2**، **1** و زیر لیست دوم، سطر دوم (سطر یک) از آرایه را با مقادیر **6** و **5**، **4** مقداردهی می‌کند. اگر براکت‌های اطراف هر زیر لیست از لیست مقداردهی اولیه **array1** حذف شود، کامپایلر مبادرت به مقداردهی اولیه عناصر سطر صفر بدنبال عناصر سطر یک می‌کند.

اعلان **array2** در خط 12 فقط حاوی پنج مقداردهی اولیه است. مقدار دهی به سطر صفر و سپس سطر یک تخصیص یافته‌اند. هر عنصری که دارای مقداردهی صریح نباشد با صفر مقداردهی اولیه می‌شود، از اینرو **array2[1][2]** با صفر مقداردهی اولیه می‌شود.

اعلان صورت گرفته برای **array3** در خط 13 سه مقداردهی در دو زیر لیست تدارک دیده است. زیر لیست برای سطر صفر بصورت صریح، دو عنصر اول از سطر صفر را با 1 و 2 مقداردهی می‌کند و عنصر سوم بصورت ضمنی با صفر مقداردهی می‌شود. زیر لیست سطر 1 بصورت صریح، اولین عنصر را با 4 و دو عنصر آخر را بصورت ضمنی با صفر مقداردهی می‌کند.

برنامه، تابع **printArray** را برای چاپ عناصر آرایه فراخوانی می‌کند. توجه کنید که در تعریف تابع (خطوط 38-27) پارامتر **const int a[][3]** مشخص شده است. زمانیکه تابع، آرایه یک بعدی را بعنوان آرگومان دریافت می‌کند، براکت‌های آرایه در لیست پارامتری تابع خالی هستند. نیازی نیست که سایز بُعد اول (یعنی تعداد سطرها) در یک آرایه دوبعدی مشخص گردد، اما به سایز بُعد، بعدی نیاز است. کامپایلر با استفاده از این سایزها موقعیت عناصر آرایه چند بُعدی در حافظه را تعیین می‌کند. تمام عناصر آرایه صرفنظر از تعداد ابعاد آرایه بصورت متوالی در حافظه ذخیره می‌شوند. در یک آرایه دو بُعدی، سطر صفرم در حافظه‌ای که بدنبال سطر یکم قرار دارد ذخیره می‌شود. در یک آرایه دو بُعدی، هر سطر، یک آرایه یک بُعدی است. برای یافتن یک عنصر در یک سطر خاص، بایستی تابع بطور دقیق از تعداد عناصر موجود در هر سطر مطلع باشد، بنابر این می‌تواند به تعداد صحیح از روی مکان‌های حافظه پرش کند. از اینرو، به هنگام دسترسی به **a[1][2]** تابع می‌داند که باید از سه عنصر سطر صفرم در حافظه پرش کند تا به سطر یکم دست یابد. سپس تابع به عنصر 2 از آن سطر دسترسی پیدا می‌کند.

در بسیاری از آرایه‌ها، از ساختار تکرار **for** برای کار با عناصر آرایه استفاده می‌شود. فرض کنید که آرایه‌ای بنام **a** موجود است. ساختار **for** تمام عناصر موجود در سطر دوم از آرایه **a** را با صفر تنظیم می‌کند:

```
for (column = 0; column < 4; column++)
    a[2][column] = 0;
```

در این عبارت، سطر سوم مشخص شده و از اینرو می‌دانیم که اولین شاخص برابر **2** خواهد بود. (**0** سطر اول و **1** سطر دوم است). حلقه **for** فقط بر روی شاخص دوم عمل می‌کند (شاخص ستون). ساختار **for** قبلی معادل عبارات تخصیصی زیر است:

```
a[2][0] = 0;
a[2][1] = 0;
a[2][2] = 0;
a[2][3] = 0;
```

ساختار تودرتوی **for** که در زیر آورده شده، مجموع تمام عناصر موجود در آرایه **a** را تعیین می‌کند:

```
total = 0;
for (row = 0; row < 3; row++)
   for (col = 0; column < 4; column++)
      total += a[row][column];
```

ساختار تودرتوی **for** مجموع تعداد عناصر در یک سطر را در هر بار محاسبه می‌کند. ساختار **for** خارجی با تنظیم شاخص **row** با **0** شروع شده و بنابراین عناصر سطر اول می‌تواند با ساختار داخلی **for**

محاسبه گردد. سپس **for** خارجی اقدام به افزایش **row** به **1** می‌کند و بنابراین سطر دوم می‌تواند محاسبه شود. ساختار خارجی **for**، مقدار **row** را به **2** افزایش می‌دهد و از اینرو سطر سوم نیز محاسبه می‌شود. زمانیکه ساختار خارجی **for** به اتمام برسد، نتایج به نمایش در می‌آیند.

۷-۱۰ مبحث آموزشی: کلاس GradeBook با استفاده از آرایه دو بعدی

در بخش ۶-۷ به معرفی کلاس **GradeBook** (برنامه‌های ۱۶-۷ و ۱۷-۷) پرداختیم که از یک آرایه، یک بعدی برای ذخیره نمرات دانشجویان در یک امتحان استفاده می‌کرد. در اکثر ترم‌ها، دانشجویان بیش از یک امتحان برگزار می‌کنند. امکان دارد اساتید مایل باشند تا نمرات یک ترم را تحلیل نمایند، هم برای دانشجو و هم برای کل کلاس.

ذخیره‌سازی نمرات دانشجویان در یک آرایه دو بعدی در کلاس GradeBook

برنامه‌های شکل ۲۳-۷ و ۲۴-۷ حاوی نسخه‌ای از کلاس **GradeBook** هستند که از یک آرایه دوبعدی **grades** برای ذخیره سازی تعداد نمرات دانشجویان در چند آزمون استفاده می‌کنند. هر سطر از آرایه نشاندهنده نمرات یک دانشجو در کل دوره بوده و هر ستون نشاندهنده تمام نمرات کل دانشجویان در یک امتحان خاص است. یک برنامه سرویس‌گیرنده همانند fig07_25.cpp آرایه‌ای را بعنوان یک آرگومان به سازنده **GradeBook** ارسال می‌کند. در این مثال، از یک آرایه 10 در 3 حاوی نمرات ده دانشجو در سه آزمون استفاده کرده‌ایم.

پنج تابع عضو (اعلان شده در خطوط 27-23 از شکل ۲۳-۷) برای کار با آرایه به منظور پردازش نمرات بکار گرفته شده‌اند. هر کدامیک از این توابع عضو مشابه توابع موجود در آرایه یک بعدی نسخه **GradeBook** هستند (برنامه‌های ۱۶-۷ و ۱۷-۷). تابع عضو **getMinimum** (تعریف شده در خطوط 65- 82 شکل ۲۴-۷) تعیین کننده کمترین نمره هر دانشجو در ترم است. تابع عضو **getMaximum** تعیین کننده بالاترین نمره هر دانشجو در ترم است (تعریف شده در خطوط 102-85 از شکل ۲۴-۷). تابع عضو **getAverage** (خطوط 115-105 از شکل ۲۴-۷) تعیین کننده میانگین یک دانشجو در طول ترم است. تابع عضو **outputBarChart** (خطوط 149-118 از شکل ۲۴-۷) نمودار میله‌ای از توزیع نمرات دانشجویان در طول ترم است. تابع عضو **outputGrades** (خطوط 177-152 از شکل ۲۴-۷) محتویات آرایه دو بعدی را در فرمت جدولی در کنار میانگین ترمی هر دانشجو چاپ می‌کند.

```
1  // Fig. 7.23: GradeBook.h
2  // Definition of class GradeBook that uses a
3  // two-dimensional array to store test grades.
4  // Member functions are defined in GradeBook.cpp
```

```
5  #include <string> // program uses C++ Standard Library string class
6  using std::string;
7
8  // GradeBook class definition
9  class GradeBook
10 {
11 public:
12    // constants
13    const static int students = 10; // number of students
14    const static int tests = 3; // number of tests
15
16    // constructor initializes course name and array of grades
17    GradeBook( string, const int [][ tests ] );
18
19    void setCourseName( string ); // function to set the course name
20    string getCourseName(); // function to retrieve the course name
21    void displayMessage(); // display a welcome message
22    void processGrades(); // perform various operations on the grade data
23    int getMinimum(); // find the minimum grade in the grade book
24    int getMaximum(); // find the maximum grade in the grade book
25    double getAverage( const int [], const int ); // find average of grades
26    void outputBarChart(); // output bar chart of grade distribution
27    void outputGrades(); // output the contents of the grades array
28 private:
29    string courseName; // course name for this grade book
30    int grades[ students ][ tests ]; // two-dimensional array of grades
31 }; // end class GradeBook
```

شکل ۲۳-۷ | تعریف کلاس GradeBook با یک آرایه دو بعدی برای ذخیره‌سازی نمرات.

```
1    // Fig. 7.24: GradeBook.cpp
2    // Member-function definitions for class GradeBook that
3    // uses a two-dimensional array to store grades.
4    #include <iostream>
5    using std::cout;
6    using std::cin;
7    using std::endl;
8    using std::fixed;
9
10   #include <iomanip> // parameterized stream manipulators
11   using std::setprecision; // sets numeric output precision
12   using std::setw; // sets field width
13
14   // include definition of class GradeBook from GradeBook.h
15   #include "GradeBook.h"
16
17   // two-argument constructor initializes courseName and grades array
18   GradeBook::GradeBook( string name, const int gradesArray[][ tests ] )
19   {
20      setCourseName( name ); // initialize courseName
21
22      // copy grades from gradeArray to grades
23      for ( int student = 0; student < students; student++ )
24
25         for ( int test = 0; test < tests; test++ )
26            grades[ student ][ test ] = gradesArray[ student ][ test ];
27   } // end two-argument GradeBook constructor
28
29   // function to set the course name
30   void GradeBook::setCourseName( string name )
31   {
32      courseName = name; // store the course name
33   } // end function setCourseName
34
35   // function to retrieve the course name
36   string GradeBook::getCourseName()
37   {
38      return courseName;
39   } // end function getCourseName
40
```

```
41  // display a welcome message to the GradeBook user
42  void GradeBook::displayMessage()
43  {
44     // this statement calls getCourseName to get the
45     // name of the course this GradeBook represents
46     cout << "Welcome to the grade book for\n" << getCourseName() << "!"
47        << endl;
48  } // end function displayMessage
49
50  // perform various operations on the data
51  void GradeBook::processGrades()
52  {
53     // output grades array
54     outputGrades();
55
56     // call functions getMinimum and getMaximum
57     cout << "\nLowest grade in the grade book is " << getMinimum()
58        << "\nHighest grade in the grade book is " << getMaximum() << endl;
59
60     // output grade distribution chart of all grades on all tests
61     outputBarChart();
62  } // end function processGrades
63
64  // find minimum grade
65  int GradeBook::getMinimum()
66  {
67     int lowGrade = 100; // assume lowest grade is 100
68
69     // loop through rows of grades array
70     for ( int student = 0; student < students; student++ )
71     {
72        // loop through columns of current row
73        for ( int test = 0; test < tests; test++ )
74        {
75           // if current grade less than lowGrade, assign it to lowGrade
76           if ( grades[ student ][ test ] < lowGrade )
77              lowGrade = grades[ student ][ test ]; // new lowest grade
78        } // end inner for
79     } // end outer for
80
81     return lowGrade; // return lowest grade
82  } // end function getMinimum
83
84  // find maximum grade
85  int GradeBook::getMaximum()
86  {
87     int highGrade = 0; // assume highest grade is 0
88
89     // loop through rows of grades array
90     for ( int student = 0; student < students; student++ )
91     {
92        // loop through columns of current row
93        for ( int test = 0; test < tests; test++ )
94        {
95         // if current grade greater than lowGrade, assign it to highGrade
96           if ( grades[ student ][ test ] > highGrade )
97              highGrade = grades[ student ][ test ]; // new highest grade
98        } // end inner for
99     } // end outer for
100
101    return highGrade; // return highest grade
102 } // end function getMaximum
103
104 // determine average grade for particular set of grades
105 double GradeBook::getAverage( const int setOfGrades[], const int grades )
106 {
107    int total = 0; // initialize total
108
109    // sum grades in array
110    for ( int grade = 0; grade < grades; grade++ )
```

```
111        total += setOfGrades[ grade ];
112
113     // return average of grades
114     return static_cast< double >( total ) / grades;
115  } // end function getAverage
116
117  // output bar chart displaying grade distribution
118  void GradeBook::outputBarChart()
119  {
120     cout << "\nOverall grade distribution:" << endl;
121
122     // stores frequency of grades in each range of 10 grades
123     const int frequencySize = 11;
124     int frequency[ frequencySize ] = { 0 };
125
126     // for each grade, increment the appropriate frequency
127     for ( int student = 0; student < students; student++ )
128
129        for ( int test = 0; test < tests; test++ )
130           ++frequency[ grades[ student ][ test ] / 10 ];
131
132     // for each grade frequency, print bar in chart
133     for ( int count = 0; count < frequencySize; count++ )
134     {
135        // output bar label ("0-9:", ..., "90-99:", "100:" )
136        if ( count == 0 )
137           cout << "  0-9: ";
138        else if ( count == 10 )
139           cout << "  100: ";
140        else
141           cout << count * 10<< "-" << ( count * 10 ) + 9 << ": ";
142
143        // print bar of asterisks
144        for ( int stars = 0; stars < frequency[ count ]; stars++ )
145           cout << '*';
146
147        cout << endl; // start a new line of output
148     } // end outer for
149  } // end function outputBarChart
150
151  // output the contents of the grades array
152  void GradeBook::outputGrades()
153  {
154     cout << "\nThe grades are:\n\n";
155     cout << "            "; // align column heads
156
157     // create a column heading for each of the tests
158     for ( int test = 0; test < tests; test++ )
159       cout << "Test " << test + 1 << "  ";
160
161     cout << "Average" << endl; // student average column heading
162
163     // create rows/columns of text representing array grades
164     for ( int student = 0; student < students; student++ )
165     {
166        cout << "Student " << setw( 2 ) << student + 1;
167
168        // output student's grades
169        for ( int test = 0; test < tests; test++ )
170           cout << setw( 8 ) << grades[ student ][ test ];
171
172        // call member function getAverage to calculate student's average;
173        // pass row of grades and the value of tests as the arguments
174        double average = getAverage( grades[ student ], tests );
175       cout << setw( 9 ) << setprecision( 2 ) << fixed << average << endl;
176     } // end outer for
177  } // end function outputGrades
```

شکل ۲۴-۷ | تابع عضو کلاس GradeBook برای کار با نمرات ذخیره شده در آرایه دو بعدی.

توابع عضو **getMinimum**، **getMaxiumu**، **outputBarChart** و **outputGrades** توسط عبارات **for** یک به یک بکار گرفته شده و بر روی آرایه **grades** اعمال می‌شوند. برای مثال به عبارت **for** تودرتو در تابع عضو **getMinimum** (خطوط 79-70) توجه کنید. عبارت **for** خارجی با تنظیم **student** با صفر شروع می‌شود (یعنی شاخص سطر)، از اینرو عناصر سطر صفر می‌توانند با متغیر **lowGrade** در بدنه عبارت **for** داخلی مقایسه گردند. حلقه عبارت **for** داخلی در میان نمرات یک سطر خاص حرکت کرده و هر نمره را با **lowGrade** مقایسه می‌کند. اگر نمره کمتر از **lowGrade** باشد، **lowGrade** را با آن نمره تنظیم می‌کند. سپس عبارت **for** خارجی مبادرت به افزایش شاخص سطر به 1 می‌کند. عناصر سطر 1 با متغیر **lowGrade** مقایسه می‌شوند. سپس عبارت **for** خارجی مقدار شاخص سطر را به 2 افزایش می‌دهد و عناصر سطر 2 با متغیر **lowGrade** مقایسه می‌شوند. این عمل تا پیمایش تمام سطرهای **grades** تکرار می‌گردد. پس از کامل شدن اجرای عبارت تودرتو، متغیر **lowGrade** حاوی کمترین نمره در آرایه دو بعدی خواهد بود. عملکرد تابع عضو **getMaximum** شبیه تابع عضو **getMinimum** است.

تابع عضو **outputBarChart** در شکل ۲۴–۷ تقریباً مشابه با تابع موجود در شکل ۱۷–۷ است. با این همه، کل نمرات یک ترم را به نمایش در می‌آورد. این تابع عضو از یک **for** تودرتو (خطوط 130-127) برای ایجاد یک آرایه یک بعدی **frequency** برپایه تمام نمرات در آرایه دو بعدی استفاده کرده است. مابقی کد در هر دو تابع عضو **outputBarChart** که نمودار را بنمایش درمی‌آورند، یکسان است. تابع عضو **outputGrades** (خطوط 177-152) هم از عبارت **for** تودرتو برای چاپ مقادیر آرایه **grades** به همراه میانگین نمرات هر دانشجو استفاده کرده است. خروجی برنامه شکل ۲۵–۷ نشاندهنده نتیجه با فرمت جدولی است. خطوط 159-158 سرآیند ستون برای هر تست را چاپ می‌کند. از یک عبارت **for** به روش شمارنده کنترل استفاده کرده‌ایم تا بتوانیم هر تست را با یک عدد تشخیص دهیم. به همین ترتیب، عبارت **for** در خطوط 176-164 برچسب اولین سطر را با استفاده از یک متغیر شمارنده برای شناسایی هر دانشجو چاپ می‌کند (خط 166).

```
1  // Fig. 7.25: fig07_25.cpp
2  // Creates GradeBook object using a two-dimensional array of grades.
3
4  #include "GradeBook.h" // GradeBook class definition
5
6  // function main begins program execution
7  int main()
8  {
9     // two-dimensional array of student grades
10    int gradesArray[ GradeBook::students ][ GradeBook::tests ] =
11       { { 87, 96, 70 },
12         { 68, 87, 90 },
13         { 94, 100, 90 },
14         { 100, 81, 82 },
15         { 83, 65, 85 },
16         { 78, 87, 65 },
17         { 85, 75, 83 },
18         { 91, 94, 100 },
```

```
19          { 76, 72, 84 },
20          { 87, 93, 73 } };
21
22     GradeBook myGradeBook(
23        "CS101 Introduction to C++ Programming", gradesArray );
24     myGradeBook.displayMessage();
25     myGradeBook.processGrades();
26     return 0; // indicates successful termination
27 } // end main
```

```
Welcome to the grade book for
CS101 Introduction to C++ Programming!

The grades are:
               Test 1   Test 2   Test 3   Average

student  1          87       96       70     84.33
student  2          68       87       90     81.67
student  3          94      100       90     94.67
student  4         100       81       82     87.67
student  5          83       65       85     77.67
student  6          78       87       65     76.67
student  7          85       75       83     81.00
student  8          91       94      100     95.00
student  9          76       72       84     77.33
student 10          87       93       73     84.33

Lowest grade in the grade book is 65
Highest grade in the grade book is 100

Overall grade distribution:
  0-9:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: ***
70-79: ******
80-89: ***********
90-99: *******
  100: ***
```

شکل ۲۵-۷ | ایجاد یک شی از کلاس GradeBook با استفاده از آرایه دو بعدی نمرات، و فراخوانی تابع عضو processGrades برای تحلیل نمرات.

اگر چه شاخص آرایه با صفر شروع می‌شود، اما دقت کنید که در خطوط 159 و 166 خروجی با عبارت **test+1** و **student+1** تنظیم می‌شود، تا شماره تست و تعداد دانشجویان از 1 شروع شود (به شکل ۲۵-۷ نگاه کنید). عبارت **for** داخلی در خطوط 170-169 از متغیر شمارنده **for** خارجی بنام **student** برای حرکت در میان یک سطر خاص از آرایه **grades** و چاپ نمره هر دانشجو استفاده کرده است. سرانجام، خط 174 میانگین هر دانشجو را با ارسال سطر جاری از **grades** (یعنی **grades[student]**)به تابع عضو **getAverage** بدست می‌آورد. تابع عضو **getAverage** (خطوط 115-105) دو آرگومان دریافت می‌کند (یک آرایه یک بعدی از نتایج آزمون برای یک دانشجو خاص و نتایج آزمون (تست) در آرایه). زمانیکه خط 174 تابع **getAverage** را فراخوانی می‌کند، آرگومان اول **grades[student]** است که مشخص

کننده یک سطر خاص از آرایه دو بعدی **grades** است که باید به **getAverage** ارسال شود. برای مثال، برپایه آرایه ایجاد شده در برنامه شکل ۲۵-۷، آرگومان **grades[1]** نشاندهنده سه مقدار (آرایه یک بعدی از نمرات) ذخیره شده در سطر 1 از آرایه دو بعدی **grades** است. می‌توان به آرایه دو بعدی بصورت آرایه‌ای که عناصر آن در آرایه‌های یک بعدی قرار دارند، نگاه کرد. تابع عضو **getAverage** مجموع عناصر آرایه را محاسبه کرده، آنرا به تعداد نتایج آزمون تقسیم و نتیجه را بصورت یک مقدار **double** برگشت می‌دهد (خط 114).

تست کلاس GradeBook

برنامه شکل ۲۵-۷ یک شی از کلاس **GradeBook** (شکل‌های ۲۳-۷ و ۲۴-۷) با استفاده از یک آرایه دو بعدی بنام **gradesArray** از نوع **int** ایجاد می‌کند (اعلان و مقداردهی شده در خطوط 20-10). توجه کنید که خط 10 به ثابت استاتیک **student** و **tests** کلاس **GradeBook** دسترسی پیدا می‌کند تا سایز هر بعد آرایه **gradesArray** بدست آید. خطوط 23-22 نام دوره و **gradesArray** را به سازنده **GradeBook** ارسال می‌کنند. سپس خطوط 25-24 توابع عضو **displayMessage** و **processGrades** را به ترتیب برای نمایش پیغام خوش‌آمدگویی و بدست آوردن گزارشی از نمرات دانشجویان در طول ترم فراخوانی می‌کنند.

۷-۱۱ معرفی کلاس استاندارد vector

در این بخش به معرفی کلاس الگوی ***vector*** (*بردار*) از کتابخانه استاندارد C++ می‌پردازیم که عرضه‌کننده نوع قدرتمندی از آرایه با قابلیت‌های بیشتر است. همانطوری که در فصل‌های بعدی کتاب و دوره‌های C++ پیشرفته مشاهده خواهیدکرد، آرایه‌های مبتنی بر اشاره‌گر در سبک C (نوع آرایه‌های معرفی شده تا بدین جا) زمینه بسیار زیادی برای تولید خطا دارند. برای مثال، همانطوری‌که قبلاً گفته شد، برنامه می‌تواند به آسانی از مرز آرایه خارج شود، چرا که C++ کنترلی بر روی شاخص‌های آرایه انجام نمی‌دهد تا جلوی خارج شدن آنها را از مرز آرایه بگیرد. دو آرایه را نمی‌توان بطور موثر با عملگرهای مقایسه‌ای یا رابطه‌ای با یکدیگر مقایسه کرد. همانطوری که در فصل ۸ خواهید آموخت، متغیرهای اشاره‌گر (که بعنوان اشاره‌گر شناخته می‌شوند) حاوی آدرس‌های حافظه بعنوان مقادیر خود هستند. اسامی آرایه‌ها اشاره‌گرهای ساده‌ای هستند که شروع آرایه در حافظه را نشان می‌دهند، و البته دو آرایه همیشه در مکان‌های مختلف حافظه جای داده می‌شوند. زمانیکه آرایه‌ای به یک تابع که برای کار با آرایه‌ها با هر سایز طراحی شده است، ارسال می‌گردد باید سایز آرایه هم در نظر گرفته شود. علاوه بر این، نمی‌توان یک آرایه را به کمک عملگر تخصیص به آرایه دیگری انتساب داد. اسامی آرایه‌ها از نوع *اشاره‌گرهای ثابت* (*const*) هستند و همانطوری که در فصل هشتم خواهید آموخت، یک ثابت اشاره‌گر نمی‌تواند در

سمت چپ یک عملگر تخصیص قرار داده شود. این قابلیت‌ها و رفتارهای دیگر به هنگام بررسی آرایه‌ها جزء ماهیت طبیعی آنها بنظر می‌رسند، اما C++ چنین قابلیت‌های را تدارک ندیده است. با این همه کتابخانه استاندارد C++ دارای کلاس الگو بنام **vector** (بردار) است که به برنامه نویسان امکان ایجاد آرایه‌های بسیار قدرتمند و با خطاهای بسیار کم را می‌دهد. در فصل ۱۱، با قابلیت‌های بسیار زیاد **vector** آشنا خواهید شد.

کلاس **vector** برای استفاده در هر برنامه C++ در اختیار است. امکان دارد نشانه‌گذاری که در مثال‌های **vector** استفاده می‌کنیم برای شما چندان آشنا نباشند، چرا که بردارها از نشانه‌گذاری الگو استفاده می‌کنند. بخش ۱۸-۶ را که در ارتباط با الگوهای تابع بود بخاطر آورید. در فصل ۱۴ در ارتباط با الگوهای کلاس صحبت خواهیم کرد. برای این بخش کافیست به گرامر بکار رفته در مثال دقت کنید.

برنامه شکل ۲۶-۷ به توصیف قابلیت‌های تدارک دیده شده توسط کلاس **vector** پرداخته که در آرایه‌های مبتنی بر اشاره‌گر سبک C وجود ندارند. کلاس **vector** از بسیاری از ویژگیهای تدارک دیده شود توسط کلاس **Array** برخوردار است که در فصل ۱۱ به ساخت آنها اقدام خواهیم کرد. کلاس **vector** در سرآیند <**vector**> تعریف شده (خط ۱۱) و متعلق به فضای نامی **std** می‌باشد (خط 12).

خطوط 20-19 دو شی **vector** برای ذخیره سازی مقادیری از نوع **int** بنام **integers1** حاوی هفت عنصر، و **integers2** حاوی 10 عنصر ایجاد کرده‌اند. بطور پیش‌فرض، تمام عناصر هر شی **vector** با صفر تنظیم می‌شوند. دقت کنید که بردارها می‌توانند برای ذخیره سازی هر نوع داده با جایگزین کردن **int** در **vector<int>** با نوع داده مقتضی تعریف شوند. این نشانه‌گذاری که تصریح کننده نوع ذخیره شده در **vector** است، مشابه نشانه‌گذاری الگوها است که در بخش ۱۸-۶ توضیح داده شده است. در فصل ۱۴ با این گرامر به تفصیل آشنا خواهید شد.

```
1  // Fig. 7.26: fig07_26.cpp
2  // Demonstrating C++ Standard Library class template vector.
3  #include <iostream>
4  using std::cout;
5  using std::cin;
6  using std::endl;
7
8  #include <iomanip>
9  using std::setw;
10
11 #include <vector>
12 using std::vector;
13
14 void outputVector( const vector< int > & ); // display the vector
15 void inputVector( vector< int > & ); // input values into the vector
16
17 int main()
18 {
19    vector< int > integers1( 7 ); // 7-element vector< int >
20    vector< int > integers2( 10 ); // 10-element vector< int >
```

```

   // print integers1 size and contents
   cout << "Size of vector integers1 is " << integers1.size()
      << "\nvector after initialization:" << endl;
   outputVector( integers1 );

   // print integers2 size and contents
   cout << "\nSize of vector integers2 is " << integers2.size()
      << "\nvector after initialization:" << endl;
   outputVector( integers2 );

   // input and print integers1 and integers2
   cout << "\nEnter 17 integers:" << endl;
   inputVector( integers1 );
   inputVector( integers2 );

   cout << "\nAfter input, the vectors contain:\n"
      << "integers1:" << endl;
   outputVector( integers1 );
   cout << "integers2:" << endl;
   outputVector( integers2 );

   // use inequality (!=) operator with vector objects
   cout << "\nEvaluating: integers1 != integers2" << endl;

   if ( integers1 != integers2 )
      cout << "integers1 and integers2 are not equal" << endl;

   // create vector integers3 using integers1 as an
   // initializer; print size and contents
   vector< int > integers3( integers1 ); // copy constructor

   cout << "\nSize of vector integers3 is " << integers3.size()
      << "\nvector after initialization:" << endl;
   outputVector( integers3 );

   // use assignment (=) operator with vector objects
   cout << "\nAssigning integers2 to integers1:" << endl;
   integers1 = integers2; // integers1 is larger than integers2

   cout << "integers1:" << endl;
   outputVector( integers1 );
   cout << "integers2:" << endl;
   outputVector( integers2 );

   // use equality (==) operator with vector objects
   cout << "\nEvaluating: integers1 == integers2" << endl;

   if ( integers1 == integers2 )
      cout << "integers1 and integers2 are equal" << endl;

   // use square brackets to create rvalue
   cout << "\nintegers1[5] is " << integers1[ 5 ];

   // use square brackets to create lvalue
   cout << "\n\nAssigning 1000 to integers1[5]" << endl;
   integers1[ 5 ] = 1000;
   cout << "integers1:" << endl;
   outputVector( integers1 );

   // attempt to use out-of-range subscript
   cout << "\nAttempt to assign 1000 to integers1.at( 15 )" << endl;
   integers1.at( 15 ) = 1000; // ERROR: out of range
   return 0;
} // end main

// output vector contents
void outputVector( const vector< int > &array )
{
   size_t i; // declare control variable
```

```
91
92    for ( i = 0; i < array.size(); i++ )
93    {
94       cout << setw( 12 ) << array[ i ];
95
96       if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
97          cout << endl;
98    } // end for
99
100    if ( i % 4 != 0 )
101       cout << endl;
102 } // end function outputVector
103
104 // input vector contents
105 void inputVector( vector< int > &array )
106 {
107    for ( size_t i = 0; i < array.size(); i++ )
108       cin >> array[ i ];
109 } // end function inputVector
```

```
Size of vector integers1 is 7
vector after initialization:
           0           0           0           0
           0           0           0

Size of vector integers2 is 10
vector after initialization:
           0           0           0           0
           0           0           0           0
           0           0

Enter 17 integers:
1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17

After input, the vectors contain:
integers1:
           1           2           3           4
           5           6           7
integers2:
           8           9          10          11
          12          13          14          15
          16          17
Evaluating: integers1 != integers2
integers1 and integeres2 are not equal

Size of vector integers3 is 7
vector after initialization:
           1           2           3           4
           5           6           7
Assigning integers2 to integers1:
inregers1:
           8           9          10          11
          12          13          14          15
          16          17
inregers2:
           8           9          10          11
          12          13          14          15
          16          17
Evaluating: integers1 == integers2
integers1 and integers2 are egual

integers1[5] is 13

assigning 1000 to integers[5]
```

```
integer1:
         8           9          10          11
         12        1000          14          15
         16          17
Attempt to assign 1000 to integers1.at( 15 )
abnormal program termination
```

شکل ۲۶-۷ | کلاس استاندارد vector.

در خط 23 از تابع عضو بردار **size** برای آوردن سایز (یعنی تعداد عناصر) **integers1** استفاده شده است. در خط 25، **integers1** به تابع **outputVector** (خطوط 102-88) ارسال شده که از براکت‌ها (**[]**) برای تهیه مقدار هر عنصر از **vector** بعنوان مقداری که بتوان برای خروجی استفاده کرد، استفاده کرده است. به شباهت این نشانه‌گذاری با نشانه‌گذاری بکار رفته برای دسترسی به مقدار یک عنصر آرایه دقت کنید. خطوط 28 و 30 همان وظایف را برای **integers2** انجام می‌دهند.

تابع عضو **size** از کلاس **vector** تعداد عناصر در یک بردار را بعنوان مقداری از نوع **size_t** برگشت می‌دهد (که نشاندهنده نوع **unsigned int** بر روی بسیاری از سیستم‌ها است). بعنوان نتیجه، خط 90 متغیر کنترلی i را از نوع **size_t** اعلان کرده است. زمانیکه شرط تکرار حلقه (خط 92) با یک مقدار **signed** (یعنی **int i**) و یک مقدار **unsigned** (یعنی مقداری از نوع **size_t** برگشتی از تابع **size**) مقایسه شود و اعلان **i** بعنوان یک **int** سبب می‌شود برخی از کامپایلرها یک پیغام هشدار صادر نماید.

خطوط 35-34 مبادرت به ارسال **integers1** و **integers2** به تابع **inputVector** (خطوط 109-105) برای خواندن مقادیر برای عناصر هر **vector** از سوی کاربر می‌کنند. تابع **inputVector** از براکت‌ها (**[]**) برای بدست آوردن مقادیر سمت چپ (*lvalues*) که می‌توانند برای ذخیره سازی مقادیر ورودی در هر عنصر **vector** بکار گرفته شوند، استفاده کرده است.

خط 46 بیان می‌کند که شی‌های **vector** می‌توانند بطور مستقیم با عملگر **=!** مقایسه کردند. اگر محتویات دو بردار برابر نباشند، عملگر مقدار **true** و در غیر اینصورت **false** برگشت می‌دهد.

کلاس **vector** به برنامه‌نویسان اجازه می‌دهد تا یک شی جدید **vector** ایجاد کنند که با محتویات **vector** موجود مقداردهی اولیه شود. در خط 51 یک شی بردار (**integers3**) ایجاد و با کپی از **integers1** مقداردهی شده است. با اینکار سازنده کپی‌کننده بردار برای انجام عملیات کپی، فعال می‌شود. در فصل ۱۱ با این نوع از سازنده‌ها بیشتر آشنا خواهید شد. خطوط 53 و 55 سایز و محتویات **integere3** را برای نشان دادن اینکه عملیات مقداردهی بدرستی صورت گرفته، به نمایش در می‌آورند.

خط 59 برای توصیف اینکه می‌توان از عملگر تخصیص (=) در شی‌های برداری استفاده کرده، مبادرت به تخصیص **integers2** به **integers1** کرده است. خطوط 62 و 64 محتویات هر دو شی را برای نمایش اینکه هر دو دارای مقادیر یکسان هستند در خروجی ظاهر کرده‌اند. سپس خط 69 مبادرت به مقایسه

integers1 با **integers2** توسط عملگر تساوی (= =) کرده تا تعیین کند که محتویات دو شی پس از انجام تخصیص در خط 59 یکسان هستند. خطوط 73 و 77 نشان می‌دهند که برنامه می‌تواند از جفت براکت‌ها (**[]**) برای بدست آوردن یک عنصر بردار بعنوان یک *lvalue* غیرقابل تغییر و بعنوان یک *lvalue* تغییرپذیر استفاده کند. یک *lvalue* تغییر نیافته عبارتی است که شی موجود در حافظه را نشان می‌دهد (همانند یک عنصر در بردار)، اما نمی‌تواند برای تغییر شی بکار گرفته شود. همچنین یک مقدار *lvalue* تغییرپذیر هم شناسه‌ای از یک شی در حافظه است، اما می‌تواند برای اصلاح و تغییر شی بکار گرفته شود. همانند آرایه‌های مبتنی بر اشاره‌گر سبک C، زبان C++ در مورد بررسی مرزها به هنگام دسترسی به عناصر بردار با استفاده از براکت‌ها هیچ تستی انجام نمی‌دهد. از اینرو، بایستی برنامه‌نویس از عملکرد عباراتی که از براکت‌ها استفاده می‌کنند مطمئن گردد تا بطور تصادفی مبادرت به دستکاری عناصری خارج از مرزهای بردار نکنند. با این همه، کلاس الگوی استاندارد **vector** دارای تابع عضو **at** است که به بررسی مرزها با به راه‌انداختن یک استثناء می‌پردازد (فصل شانزدهم). اگر آرگومان این تابع یک شاخص معتبر نباشد، یک استثناء بوجود می‌آورد. بطور پیش‌فرض این حالت سبب خاتمه یافتن برنامه C++ می‌شود. اگر شاخص معتبر باشد، تابع **at** مبادرت به برگشت دادن آن از آن مکان بعنوان یک *lvalue* تغییرپذیر یا غیرقابل تغییر براساس زمینه (غیرثابت یا ثابت) در فراخوانی صورت گرفته می‌کند. در خط 83 نمونه‌ای از فراخوانی تابع با یک شاخص غیرمعتبر نشان داده شده است.

۱۲-۷ مبحث آموزشی مهندسی نرم‌افزار: همکاری مابین شی‌های سیستم ATM

در این بخش، به بررسی همکاری (تعاملات) صورت گرفته مابین شی‌های موجود در سیستم ATM می‌پردازیم. زمانیکه دو شی برای انجام دادن یک وظیفه با یکدیگر ارتباط برقرار می‌کنند، گفته می‌شود که باهم همکاری می‌نمایند و اینکار با فعال کردن عملیات‌ها در سمت مقابل صورت می‌گردد. همکاری متشکل از ارسال یک پیغام از سوی شی از یک کلاس به شی از یک کلاس دیگر است. پیغام‌ها در C++ از طریق فراخوانی تابع عضو ارسال می‌شوند.

در بخش ۱۸-۶ به بررسی تعدادی از عملیات‌ها که در کلاس‌های سیستم ATM رخ می‌دادند پرداختیم. در این بخش، تمرکز ما بر روی پیغام‌های است که این عملیات‌ها را فعال می‌سازند. برای شناسایی همکاری‌های صورت گرفته در سیستم به مستند نیازها در بخش ۸-۲ مراجعه می‌کنیم. بخاطر دارید که این مستند تعیین‌کننده محدوده فعالیت‌های است که در طول مدت زمان یک جلسه **ATM** اتفاق می‌افتند (همانند تایید کاربر، انجام تراکنش‌های لازم). اولین کار ما در شناسایی همکاری‌های صورت گرفته در

سیستم، بدست آوردن توصیفی از نحوه انجام این وظایف است. همانطوری که اینکار را ادامه می‌دهیم، در مابقی بخش‌های «مهندسی نرم‌افزار» به همکاری‌های بیشتری دست خواهیم یافت.

شناسایی همکاری‌ها در سیستم

برای شناسایی همکاری‌های موجود در سیستم بدقت شروع به مطالعه بخش‌های مربوط به مستند نیازها که تصریح‌کننده آنچه که با **ATM** در تایید کاربر و انجام هر نوع تراکنش صورت دهد می‌کنیم. برای هر عمل یا مرحله توضیح داده شده در مستند نیازها، تصمیم می‌گیریم که کدام شی‌ها در سیستم باید برای دست یافتن به نتایج دلخواه در تعامل قرار داده شوند. یک شی را بعنوان شی ارسال‌کننده (شی که پیغام ارسال می‌کند) و دیگری را بعنوان شی دریافت‌کننده مشخص می‌کنیم (شی که به سرویس‌گیرنده‌های کلاس پیشنهاد عملیاتی را می‌کند). سپس یکی از عملیات‌های شی دریافت‌کننده را انتخاب می‌کنیم (مشخص شده در بخش ۱۸-۶) که باید توسط شی ارسال‌کننده برای انجام یک رفتار مناسب فعال شود. برای مثال، **ATM** در زمان بیکاری پیغام خوش‌آمدگویی را به نمایش در می‌آورد. می‌دانیم که شی از کلاس **Screen** پیغامی را به کاربر از طریق عملیات **displayMessage** خود به نمایش در می‌آورد. از اینرو، تصمیم می‌گیریم که سیستم قادر به نمایش پیغام خوش‌آمدگویی از طریق یک همکاری مابین **ATM** و **Screen** باشد به نحوی که **ATM** پیغام **diplayMessage** را به **Screen** از طریق فعال کردن عملیات **displayMessage** از کلاس **Screen** ارسال می‌کند.

در جدول ۲۷-۷ لیست همکاری‌های که می‌توان از مستند نیازها بدست آورد مشخص شده‌اند. برای هر شی ارسال‌کننده، براساس ترتیب موجود در مستند نیازها، اقدام به لیست همکاری‌ها کرده‌ایم. هر همکاری را با یک فرستنده منحصر بفرد، پیغام و دریافت‌کننده و فقط یکبار لیست کرده‌ایم، حتی اگر همکاری چندین بار در یک جلسه ATM رخ دهد. برای مثال در سطر اول جدول ۲۷-۷ مشخص است که همکاری **ATM** با **Screen** هر زمانیکه ATM نیاز بنمایش پیغامی به کاربر داشته باشد رخ می‌دهند.

شی از کلاس	ارسال پیغام	با شی از کلاس
ATM	displayMessage	Screen
	getInput	keypad
	authenticateUser	BankDatabase
	execute	BalanceInquiry
	execute	Withdrawal
	execute	Deposit
BalanceInquiry	getAvailableBalance	BankDatabase
	getTotalBalance	BankDatabase
	displayMessage	Sereen
Withdrawal	displayMessage	Screen
	getInput	Keypad
	getAvailableBalance	BankDatabase
	isSufficientCashAvailable	CashDispenser
	debit	BankDatabase
	dispenseCash	CashDrspenser

Deposit	**displyMessage**	**Screen**
	getInput	**Keypad**
	isEnvelopeReceived	**DepositSlot**
	credit	**BankDatabase**
BankDatabase	**validatePIN**	**Account**
	getAvailableBalance	**Account**
	getTotalBalance	**Account**
	debit	**Account**
	credit	**Account**

شکل ۲۷-۷ | همکاری‌های موجود در سیستم *ATM*.

اجازه دهید به بررسی همکاری‌های موجود در جدول ۲۷-۷ بپردازیم. قبل از اینکه کاربر اجازه هر نوع تعاملی را بدست آورده باشد، بایستی **ATM** به کاربر اعلان کند تا شماره حساب، سپس **PIN** خود را وارد سازد. هر کدامیک از این وظایف با ارسال پیغام **displayMessage** به **Screen** صورت می‌گیرند. هر دو این اعمال اشاره به همان همکاری مابین **ATM** و **Screen** دارند که در جدول ۲۷-۷ لیست شده‌اند. **ATM** ورودی کاربر را در واکنش به اعلان و از طریق ارسال پیغام **getInput** به صفحه‌کلید **(Keypad)** بدست می‌آورد. سپس باید ATM تعیین کند که آیا شماره حساب و **PIN** مشخص شده از سوی کاربر مطابق با حسابی در پایگاه داده است یا خیر. اینکار با ارسال پیغام **authenticateUser** (اعتبارسنجی کاربر) به **BankDatabase** یا پایگاه‌داده صورت می‌گیرد. بخاطر دارید که **BankDatabase** بطور مستقیم قادر به اعتبار سنجی کاربر نیست و فقط حساب **(Account)** کاربر می‌تواند به **PIN** کاربر برای تایید وی دسترسی داشته باشد. بنابر این در جدول ۲۷-۷ یک همکاری لیست شده که در آن **BankDatabase** مبادرت به ارسال پیغام **validatePIN** به یک **Account** می‌کند.

پس از تایید کاربر، **ATM** منوی اصلی را با ارسال دنباله‌ای از پیغام‌های **displayMessage** به **Screen** و بدست آوردن ورودی از منوی انتخابی با ارسال پیغام **getInput** به **Keypad** به نمایش در می‌آورد. همکاری‌های صورت گرفته تا بدین مرحله ذکر کرده‌ایم. پس از انتخاب نوع تراکنش مورد نظر از سوی کاربر، **ATM** مبادرت به اجرای تراکنش با ارسال پیغام **execute** به شی مناسب می‌کند (مثلاً به یک **BalanceInquiry**، **Withdrawal** یا یک **Deposit**). برای مثال، اگر کاربر تقاضای میزان موجودی را کند، ATM یک پیغام **execute** به **BalanceInquiry** ارسال خواهد کرد.

با بررسی دقیق‌تر مستند نیازها، همکاری‌های صورت گرفته در میان هر نوع تراکنش آشکارتر می‌شود. یک **BalanceInquiry** میزان پول موجود در حساب کاربر را با ارسال پیغام **getAvailableBalance** به **BalanceInquiry**، که به ارسال پیغام **getAvailableBalance** به حساب کاربر **(Account)** واکنش نشان می‌دهد، بدست می‌آورد. به همین ترتیب **BalanceInquiry** میزان پول موجود در یک سپرده را با ارسال پیغام **getTotalBalance** به **BankDatabase** بدست می‌آورد، که همان پیغام را به **Account**

کاربر ارسال می‌کند. برای نمایش هر دو مقدار از میزان موجودی کاربر در یک زمان، **BalanceInquiry** پیغام **displayMessage** را به **Screen** ارسال می‌کند.

یک **Withdrawal** (برداشت پول) دنباله‌ای از پیغام‌های **displayMessage** را به **Screen** برای نمایش یک منو از میزان پرداخت‌های استاندارد (مثلاً 20$، 40$، 60$، 100$، 200$) ارسال می‌کند. **Withdrawal** اقدام به ارسال پیغام **getInput** به **Keypad** می‌کند تا انتخاب کاربر از منو را بدست آورد. سپس تعیین می‌کند که آیا تقاضای میزان برداشت کمتر یا معادل میزان موجودی کاربر است یا خیر. **Withdrawal** می‌تواند میزان پول موجود در حساب کاربر را با ارسال پیغام **getAvailableBalance** به **BankDatabase** بدست آورد. سپس **Withdrawal** تست می‌کند که آیا پرداخت‌کننده پول (جعبه پول) به میزان کافی پول نقد در خود دارد یا خیر و اینکار را با ارسال پیغام **isSufficientCashAvailable** به **CashDispenser** انجام می‌دهد. **Withdrawal** پیغام **debit** را به **BankDatabase** ارسال می‌کند تا از میزان موجودی کاربر کاسته شود. بخاطر دارید که بدهکار کردن حساب هم در **totalBalance** و **availableBalance** اتفاق می‌افتد. برای پرداخت میزان پول درخواستی، **Withdrawal** پیغام **dispenseCash** را به **CashDispenser** ارسال می‌کند. سرانجام، **Withdrawal** پیغام **displayMessage** را **Screen** ارسال می‌کند تا به کاربر برداشت پول را یادآوری کند.

Deposit به پیغام **execute** ابتدا با ارسال یک پیغام **displayMessage** به **Screen** برای اعلان میزان سپرده‌گذاری از سوی کاربر واکنش نشان می‌دهد. **Deposit** پیغام **getInput** را به **Keypad** برای تهیه ورودی کاربر ارسال می‌کند. سپس پیغام **displayMessage** را به **Screen** می‌فرستد تا به کاربر اعلان کند که پاکت سپرده‌گذاری را وارد سازد. برای تعیین اینکه آیا شکاف سپرده‌گذاری پاکت سپرده را دریافت کرده است یا خیر، **Deposit** پیغام **isEnvelopeReceived** را به **DepositSlot** ارسال می‌نماید. **Deposit** اقدام به، به روز کردن حساب کاربر با ارسال پیغام **credit** به **BankDatabase** می‌کند و متعاقب آن یک پیغام **credit** به حساب کاربر ارسال می‌شود. بخاطر دارید که اعتبار افزوده شده به حساب فقط موجب افزایش میزان **totalBalance** می‌شود و تأثیری در **availableBalance** ندارد.

دیاگرام‌های تراکنش

اکنون که مجموعه‌ای از همکاری‌های ممکنه مابین شی‌های موجود در سیستم ATM خود را شناسایی کرده‌ایم، اجازه دهید تا این تراکنش‌ها را با استفاده از UML بصورت گرافیکی مدل سازی کنیم. زبان UML دارای چندین نوع دیاگرام تراکنشی است که رفتار یک سیستم را با مدل کردن نحوه تعامل شی‌ها با شی دیگری مدل سازی می‌کنند. تاکید *دیاگرام ارتباطی* بر مشارکت شی‌ها در همکاری‌ها است[نکته در نسخه‌های اولیه UML به دیاگرام‌های ارتباطی، *دیاگرام‌های همکاری* گفته می‌شود.] همانند دیاگرام

ارتباطی، *دیاگرام توالی* نمایشی از همکاری‌ها در میان شی‌ها است، اما با تاکید بر *زمان ارسال* پیغام‌ها مابین شی‌ها.

دیاگرام‌های ارتباطی

شکل ۲۸-۷ نمایشی از یک دیاگرام ارتباطی است که اجرای **BalanceInquiry** توسط **ATM** را مدل کرده است. شی‌های مدل شده در UML بصورت مستطیل‌های حاوی اسامی بفرم ***نام کلاس: نام شی*** نشان داده می‌شوند. در این مثال، که فقط یک شی از هر نوع باهم درگیر شده‌اند، ما نام شی را نادیده گرفته‌ایم و فقط یک کولن قبل از نام کلاس قرار داده‌ایم. [نکته: مشخص کردن نام هر شی در یک دیاگرام ارتباطی در زمان مدل کردن چندین شی از یک نوع توصیه شده است.] شی‌های ارتباطی با خطوط یک پارچه به هم متصل شده، و جهت پیغام‌های ارسالی مابین شی‌ها در امتداد این خطوط با فلش نشان داده می‌شوند. نام پیغام که در کنار فلش ظاهر می‌شود، نام یک عملیات (تابع عضو) متعلق به شی دریافت‌کننده است. می‌توانید در مورد نام همانند سرویسی فکر کنید که شی دریافت‌کننده آنرا برای شی ارسال‌کننده تدارک می‌بیند (سرویس‌گیرنده‌های خود).

شکل ۲۸-۷ | دیاگرام ارتباطی از اجرای پرس‌وجوی میزان موجودی توسط *ATM*.

فلش یکپارچه بکار رفته در شکل ۲۸-۷ نشاندهنده یک پیغام یا فراخوانی همزمان در UML و فراخوانی یک تابع در C++ است. این فلش بر این نکته دلالت دارد که جریان کنترل از سوی شی فرستنده (در اینجا **ATM**) به شی دریافت‌کننده (**BalanceInquiry**) است. از آنجا که این یک فراخوانی همزمان است، امکان ارسال یک پیغام دیگر توسط شی ارسال کننده یا انجام کاری دیگری وجود ندارد تا اینکه شی دریافت‌کننده این پیغام را پردازش کرده و کنترل به شی فرستنده برگشت داده شود. فرستنده فقط در انتظار باقی می‌ماند. برای مثال در شکل ۲۸-۷، **ATM** مبادرت به فراخوانی تابع عضو **execute** از **BalanceInquiry** می‌کند و تا زمانیکه **execute** کار خود را تمام نکرده و کنترل را به **ATM** برگشت ندارد نمی‌تواند پیغام دیگری ارسال کند. [نکته: اگر فراخوانی از نوع غیرهمزمان یا اسنکرون باشد، اینحالت با یک فلش دوطرفه نشان داده می‌شود و دیگر شی ارسال‌کننده مجبور نبود تا در انتظار برگشت کنترل از سوی شی دریافت‌کننده باقی بماند و می‌تواند به ارسال پیغام‌های دیگری بلافاصله پس از فراخوانی اسنکرون ادامه دهد. غالباً فراخوانی‌های اسنکرون در C++ با استفاده از پلات‌فرم خاصی از کتابخانه‌های تدارک دیده شده توسط کامپایلر پیاده‌سازی می‌شوند. بررسی چنین تکنیک‌های خارج از قلمرو این کتاب هستند.]

توالی پیغام‌ها در دیاگرام ارتباطی

در شکل ۲۹-۷ یک دیاگرام ارتباطی نشان داده شده است که تراکنش‌های مابین شی‌های موجود در سیستم را در زمانیکه یک شی از کلاس **BalanceInquiry** اجرا می‌شود را مدل کرده است. فرض می‌کنیم که صفت **accountNumber** شی حاوی شماره حساب کاربر جاری است. همکاری‌های موجود در شکل ۲۹-۷ پس از ارسال یک پیغام **execute** از سوی **ATM** به یک **BalanceInquiry** (یعنی تراکنش مدل شده در شکل ۲۸-۷) شروع می‌شوند.

شکل ۲۹-۷ | دیاگرام ارتباطی برای اجرای پرس وجوی میزان موجودی.

اعداد قرار گرفته در سمت چپ نام یک پیغام بر ترتیب ارسال و گذر پیغام دلالت دارند. توالی پیغام‌ها در یک دیاگرام ارتباطی دارای ترتیب عددی از کوچکترین به سمت بزرگترین عدد است. در این دیاگرام، عدد گذاری پیغام با 1 شروع و با پیغام 3 خاتمه یافته است. ابتدا **BalanceInquiry** یک پیغام **getAvailableBalance**به **BankDatabase** ارسال می‌کند (پیغام 1)، سپس پیغام **getTotalBalance** را به **BankDatabase** ارسال می‌کند (پیغام 2). در درون پرانتزهای قرار گرفته در مقابل نام پیغام، می‌توانیم یک لیست جدا شده با کاما از اسامی پارامترهای ارسالی به همراه پیغام قرار دهیم (آرگومان در فراخوانی یک تابع ++C). در این مورد **BalanceInquiry** صفت **accountNumber** را به همراه پیغام خود به **BankDatabase** ارسال می‌کند تا نشان دهد اطلاعات کدام حساب باید بازیابی شود. از شکل ۳۳-۶ بخاطر دارید که عملیات **getAvailaldeBalance** و **getTotoalBalance** از کلاس **BankDatadase** هر یک مستلزم یک پارامتر برای شناسایی حساب هستند. سپس **BalanceInquiry** مبادرت به نمایش موجودی قابل برداشت وکل موجودی با ارسال یک پیغام **displayMessage** به **Screen** (پیغام 3) به کاربر می‌کند که شامل یک پارامتر به نشانه پیغام قابل نمایش است. دقت کنید با وجود اینکه در شکل ۲۹-۷ دو پیغام از **BankDatabase** به یک حساب (**Account**) ارسال شده (پیغام‌های1و 2)، بایستی **BankDatabase** یک پیغام **getAvailableBalance** و یک پیغام **getTotalBalance** به حساب کاربر ارسال کند. به چنین پیغام‌های که در درون پیغام دیگری ارسال می‌شوند، پیغام‌های تودرتو یا آشیانه‌ای گفته می‌شود. توصیه UML بر استفاده از شماره‌گذاری اعشاری برای نشان دادن پیغام‌های تودرتو است. برای مثال، پیغام 1.1 اولین پیغام تودرتو در پیغام 1 است، **BankDatabase** یک پیغام **getAvailableBalance** را در زمانیکه **BankDatabase** در حال پردازش پیغام از همان نام است ارسال کرده است.[نکته: اگر **BankDatabase** نیازمند ارسال پیغام تودرتوی دومی باشد در حالیکه پیغام 1 پردازش می‌شود، پیغام دوم بصورت 1.2 شماره‌گذاری می‌شود.] امکان دارد یک پیغام زمانی ارسال شود که کلیه پیغام‌های تودرتو از پیغام قبلی ارسال شده باشند. برای مثال، **BalanceInquiry** پیغام 3 را فقط پس از پیغام 2 و 2.1 ارسال می‌کند.

طرح شماره‌گذاری تودرتوی بکار رفته در دیاگرام‌های ارتباطی سبب افزایش وضوح نحوه و ترتیب ارسال هر پیغام می‌شود. برای مثال، اگر پیغام‌های بکار رفته در شکل ۲۹-۷ را با استفاده از طرح عددگذاری ساده (همانند 1,2,3,4,5) شماره‌گذاری کنیم، احتمال دارد شخصی که به دیاگرام نگاه می‌کند قادر به تعیین اینکه **BankDatabase** مبادرت به ارسال پیغام **getAvailaldeBalance** (پیغام 1.1) به یک **Account** در زمانیکه **BankDatabase** در حال پردازش پیغام 1 است یا پس از کامل شدن پردازش پیغام 1 نباشد. اما شماره‌گذاری تودرتو کمک می‌کند که دومین پیغام **getAvailableBalance** (پیغام 1.1) به یک **Account** در درون اولین پیغام **getAvailableBalance** (پیغام 1) توسط **BankDatabase** ارسال شده است.

دیاگرام‌های توالی

تاکید دیاگرام‌های ارتباطی بر همکاری‌های موجود است، اما مدل‌سازی زمان در آنها چندان قوی نیست. *دیاگرام توالی* در مدل کردن زمانبندی همکاری‌ها از وضوح بیشتری برخوردار است. در شکل۳۰-۷ نمایشی از یک دیاگرام توالی که تراکنش‌های رخ داده در زمان برداشت پول **Withdrawal** را مدل کرده، آورده شده است. خطوط خط چین که از مستطیل یک شی به سمت پایین امتداد یافته‌اند، نشاندهنده خط عمر و زمان آن شی هستند. ترتیب زمانی وقوع عمل یا فرآیندی در طول عمر یک شی از بالا به سمت پایین است، عملی که در مراتب بالا قرار دارد قبل از عملی که در پایین تر از آن جای گرفته اتفاق می‌افتد.

شکل ۳۰-۷ | دیاگرام توالی مدل کننده عملیات برداشت پول *(Withdrawal)*.

ارسال پیغام در دیاگرام‌های توالی همانند ارسال پیغام در دیاگرام‌های ارتباطی است. فلش یک پارچه بسط یافته از سمت شی ارسال کننده به سمت شی دریافت‌کننده نشاندهنده یک پیغام مابین دو شی است. نوک فلش به سمت یک فعالیت در خط عمر شی دریافت‌کننده است. یک فعالیت به شکل یک مستطیل نازک عمودی نشان داده می‌شود، که نشاندهنده یک شی در حال اجرا است. زمانیکه یک شی کنترل را باز می‌گرداند، پیغام برگشتی توسط یک خط چین با فلش از سوی شی فعال که کنترل را به فرستنده پیغام بازمی‌گرداند نشان داده می‌شود. برای حذف موارد سردرگم کننده، فلش‌های پیغام برگشتی را حذف کرده‌ایم و UML به منظور افزایش خوانایی دیاگرام چنین اجازه‌ای را می‌دهد. همانند دیاگرام‌های ارتباطی، دیاگرام توالی می‌توانند نشاندهنده پارامترهای پیغام در میان پرانتزها پس از نام پیغام باشند.

توالی پیغام در شکل ۳۰-۷ زمانی شروع می‌شود که یک **Withdrawal** به کاربر اعلان می‌کند تا میزان پول مورد نظر را انتخاب کند و اینکار با ارسال پیغام **displayMessage** به **Screen** صورت می‌گیرد. سپس **Withdrawal** پیغام **getInput** را به **Keypad** می‌فرستد تا ورودی کاربر را دریافت کند. در شکل ۲۸-۵ منطق اعمال شده در فعالیت برداشت پول مدل سازی شده است و از اینرو این منطق را در این

دیاگرام توالی نشان نداده‌آیم و بجای آن بهترین سناریور را که در آن موجودی حساب کاربر بیشتر یا برابر میزان برداشتی است و پرداخت کننده پول، حاوی مقدار کافی پول نقد برای برآورده کردن تقاضا است را مدل کرده‌ایم.

پس از بدست آوردن میزان پول برای برداشت، **Withdrawal** پیغام **getAvailableBalance** را به **BankDatabase** ارسال می‌کند که آن هم در ادامه پیغام **getAvailableBalance** را به حساب کاربر (**Account**) ارسال می‌نماید. فرض کنید که حساب کاربر به میزان کافی پول دارد و می‌تواند تراکنش درخواست شده را انجام دهد، سپس **Withdrawal** پیغام **isSufficientCashAvailable** را به **CashDispenser** ارسال می‌نماید. فرض کنید که پول نقد به میزان کافی در اختیار است، **Withdrawal** موجودی کاربر را از حساب آن کم می‌کند (هم از **totalBalance** و هم از **availableBalance**) با ارسال پیغام **debit** به **BankDatabase** . پایگاه داده با ارسال پیغام **debit** به حساب کاربر (**Account**) از خود واکنش نشان می‌دهد. سرانجام **Withdrawal** پیغام **dispenseCash** را به **CashDispenser** و پیغام **displayMessage** را به **Screen** ارسال می‌کند تا به کاربر اعلان کند، پول خود را از ماشین بردارد.

تمرینات خودآزمایی مبحث مهندسی نرم‌افزار

۷-۱ ..... متشکل از ارسال یک پیغام از سوی شی از یک کلاس به شی از کلاس دیگر است.

a) وابستگی

b) اجتماع

c) همکاری

d) ترکیب

۷-۲ کدام فرم از دیاگرام تراکنشی تاکید برای نوع همکاری دارد؟ و کدامیک تاکید بر زمان همکاری؟

۷-۳ یک دیاگرام توالی ایجاد کنید که تراکنش‌های موجود مابین در زمان اجرای موفقیت آمیزی سپرده‌گذاری (Deposit) را مدل‌سازی کند.

پاسخ خودآزمایی مبحث مهندسی نرم‌افزار

۷-۱ c.

۷-۲ دیاگرام‌های ارتباطی بر نوع همکاری و دیاگرام‌های توالی بر زمان رخ دادن همکاری‌ها تاکید دارند.

شکل ۷-۳۱ دیاگرام توالی که اجرای سپرده‌گذاری (Deposit) را مدل کرده است.

خودآزمایی

۷-۱ جاهای خالی را در عبارات زیر با کلمات مناسب پر کنید:

a) مقادیر لیست‌ها و جداول می‌توانند در ..... و ..... ذخیره شوند.

b) عناصر یک آرایه دارای ..... و ..... یکسان هستند.

c) عددی که به یک عنصر آرایه اشاره می‌کند، ..... نام دارد.

d) باید از یک ..... در اعلان سایز آرایه استفاده کرد، چراکه با اینکار برنامه پایدارتر می‌شود.

e) فرآیند قراردادن عناصر یک آرایه در یک ترتیب، ..... آرایه نامیده می‌شود.

f) عمل تعیین اینکه آیا آرایه‌ای حاوی یک مقدار مشخص است، ..... نامیده می‌شود.

g) به آرایه‌های که دو یا بیش از دو شاخص دارند، آرایه‌های، ..... گفته می‌شود.

۷-۲ کدامیک از عبارات زیر صحیح و کدامیک اشتباه است. اگر عبارتی اشتباه است علت آنرا توضیح دهید.

a) یک آرایه می‌تواند مقادیری از نوع‌های مختلف در خود ذخیره سازد.

b) شاخص یک آرایه بایستی از نوع داده float باشد.

c) اگر لیست مقداردهی کننده اولیه کمتر از تعداد عناصر در آرایه باشد، مابقی عناصر با آخرین مقدار در لیست مقداردهی اولیه، مقدار دریافت خواهند کرد.

d) اگر لیست مقداردهی کننده اولیه حاوی مقادیری بیش از تعداد عناصر موجود در آرایه باشد، خطا رخ خواهد داد.

e) یک عنصر مجزا در آرایه که به تابعی ارسال شده و تغییر یافته پس از کامل شدن وظیفه تابع، حاوی مقدار تغییر یافته خواهد بود.

۷-۳ عبارتی بنویسید که موارد خواسته شده در زیر را برآورده سازد (برای آرایه‌ای بنام fractions):

a) یک متغیر ثابت نام arraySize تعریف و با 10 مقداردهی کنید.

b) آرایه‌ای با عناصر arraySize از نوع double اعلان، و آنها را با صفر مقداردهی کنید.

c) چهارمین عنصر در آرایه.

d) مراجعه به عنصر 4 آرایه.

e) تخصیص مقدار 1.667 به عنصر ۹ آرایه.

f) تخصیص مقدار 3.333 به هفتمین عنصر آرایه.

g) چاپ عناصر 6 و 9 آرایه با دقت دو رقم در سمت راست نقطه اعشار، و نمایش خروجی.

h) چاپ تمام عناصر آرایه توسط عبارت for. متغیر i را بعنوان متغیر کنترلی حلقه for تعریف کرده خروجی را بنمایش در آورید.

۷-۴ به سوالات زیر با توجه به آرایه‌ای بنام table پاسخ دهید:

a) اعلان آرایه از نوع صحیح با 3 سطر و 3 ستون. فرض کنید که متغیر ثابت arraySize با مقدار3 تعریف شده.

b) برنامه‌ای بنویسید که تا مقادیر هر عنصر آرایه table را در فرمت جدولی 3 سطر و 3 ستون چاپ کند. با فرض اینکه آرایه بصورت زیر مقداردهی اولیه شده است:

```
int table[arraySize][arraySize]={{1,8},{2,4,6},{5}};
```

و متغیرهای کنترلی i, j تعریف شده باشند. خروجی را بنمایش در آورید.

۷-۵ خطای موجود در عبارات زیر را یافته و اصلاح کنید.

a) # include < iostream>;

b) arraySize=10; // arraySsize was declared const

c) int b[10]={0};با فرض اینکه

```
for (int i=0; i<=10; i++)
    b[i]=1;
```

d) بافرض a[2][2]={{1,2},{3,4}};

```
a[i,j]=5;
```

پاسخ خودآزمایی

۷-۱ a) آرایه‌ها، بردارها. b) نام، نوع. c) شاخص یا ساب‌اسکریپت. d) متغیر ثابت. e) مرتب‌سازی. f) جستجو. g) دوبعدی.

g)نامنظم. i) const j) IndexOutOfRangeException

۷-۲ a) اشتباه. یک آرایه فقط قادر به نگهداری مقادیر از یک نوع است. b) اشتباه. شاخص آرایه باید یک مقدار یا عبارت صحیح باشد. c) اشتباه. عناصر باقیمانده با صفر مقداردهی اولیه می‌شوند.d) صحیح.e) اشتباه. عناصر مجزای آرایه به روش مقدار ارسال می‌شوند.

۷-۳

```
a) const int arraySize = 10;
b) double fractions[arraysize] = {0,0};
c) fractions[3]
d) fractions[4]
e) fractions[9] = 1.667'
f) fractions[6] = 3.333;
g) cout << fixed<<setprecision (2);
   cout << fractions[6] <<' ' << fractions[9] << endl;
```

خروجی:3.33 1.67

```
h) for (int i=0; i< arraySize;i++)
    cout <<" fractions["<<i<<"] = << fractions[i] << endl;
```

خروجی:

```
Fractions[0]=0.0
Fractions[1]=0.0
Fractions[2]=0.0
Fractions[3]=0.0
Fractions[4]=0.0
Fractions[5]=0.0
```

```
Fractions[6]=3.333
Fractions[7]=0.0
Fractions[8]=0.0
Fractions[9]=1.667
```

۷-۴

```
a)int table[arraySize][arraySize];
b)4
c) for (i=0;i<< arraySize;i++)
     for (j=0; j< arraySize; j++)
       table[i][j]=i+j;
d) cout <<"[0] [1] [2]" <<end;
   for (int i=0; i< arraySize; i++){
    cout << '[' <<i<<"]';
    for (int j=0; j< arraySize; j++)
     cout <<setw(3)<<table[i][0]<<" ";
   cout << endl;
```

خروجی:

```
    [0]  [1]  [2]
[0]  1    8    0
[1]  2    4    6
[2]  5    0    0
```

۷-۵

a) خطا: سیملوکن در انتهای #include قرار گرفته است.

اصلاح: حذف سیمکولن.

b) خطا: تخصیص مقداری به متغیر ثابت با استفاده از عبارت تخصیصی.

اصلاح: مقداردهی اولیه متغیر ثابت در const int arraySize.

c) خطا: مراجعه به عنصر آرایه خارج از مرزهای آرایه (b[10])

اصلاح: تغییر مقدار نهایی متغیر کنترلی به 9.

d) خطا: شاخص عملکرد درستی ندارد.

اصلاح: تغییر عبارت به a[1] [1]=5;.

تمرینات

۷-۶ جاهای خالی را با عبارت مناسب پر کنید.

a) اسامی چهار عنصر آرایه p (یعنی int p[1];) عبارتند از ..... ، ..... ،..... و.....

b) نامگذاری آرایه، نوع آرایه مشخص کردن تعداد عناصر و در آرایه، .....آرایه نامیده میشود.

c) بطور قرار دادی، اولین شاخص در آرایه دو بعدی بعنوان عنصر ..... و دومین شاخص بعنوان عنصر ..... شناخته می‌شود.

d) یک آرایه m در n حاوی ..... سطر، ..... ستون و ..... عنصر است.

e) نام عنصر در سطر 3 و ستون 5 آرایه d عبارت است از .....

۷-۷ تعیین کنید کدامیک از موارد زیر صحیح و کدامیک اشتباه است.

a) برای مراجعه به یک موقعیت خاص یاا عنصری در درون آرایه، نام آرایه و مقدار دقیق آن عنصر را مشخص می‌کنیم.

b) با اعلان آرایه فضا برای آرایه رزرو می‌شود.

c) برای اینکه 100 مکان برای آرایه صحیحی بنام p رزرو شود؛ باید برنامه نویس در اعلان بنویسد p[100];

d) برای مقداردهی اولیه یک آرایه 15 عنصری با صفر، باید از یک عبارت for استفاده کرد.

e)برای محاسبه مجموع عناصر یک آرایه دوبعدی باید ازعبارت for تودرتو استفاده کرد.

۷-۸ عباراتی در c++ بنویسید که موارد خواسته در زیر را بر آورده سازند:

a) نمایش مقدار عنصر 6 از آرایه کاراکتری f.

b) وارد کردن مقداری به عنصر ۴، آرایه یک بعدی اعشار، بنام b.

c) مقداردهی اولیه پنج عنصر آرایه یک بعدی g با 8.

d) محاسبه مجموع و چاپ عناصر آرایه c که 100 عنصر دارد.

e) کپی کردن آرایه a به اولین بخش آرایه b. با فرض b[34],a[11];

f) کوچکترین و بزرگترین مقدار موجود در آرایه w با 99 عنصر را یافته و چاپ کنید.

۷-۹ با در نظر گرفتن آرایه 2 در 3 بنام t و از نوع صحیح موارد خواسته شده زیر را برآورده سازید.

a) اعلانی برای t بنویسید.

b) آرایه t دارای چند ستون است؟

c) آرایه t دارای چند سطر است؟

d) آرایه t چند عنصر دارد؟

e) اسامی تمام عناصر موجود در سطر 1 از t را بنویسید.

f) اسامی تمام عناصر موجود در ستون 2 از t را بنویسید.

g) عبارتی بنویسید که عنصر قرار گرفته در سطر 1 و ستون 2 را با صفر تنظیم کند.

h) دنباله ای از دستورات بنویسید که هر عنصر از t را با صفر مقداردهی اولیه کند. از حلقه استفاده نکنید.

i) یک عبارت for تودرتو بنویسید که هر عنصر t را با صفر مقداردهی کند.

j) عبارتی بنویسید که مقادیری برای عناصر t از ترمینال دریافت کند.
k) دنباله‌ای از عبارات بنویسید که کوچکترین مقدار در آرایه t را یافته و چاپ کند.
l) عبارتی بنویسید که عناصر موجود در سطر صفرt را بنمایش در آورد.
m) عبارتی بنویسید که مجموع عناصر در ستون سوم t را بنمایش در آورد.
n) عباراتی بنویسید که آرایه t را بصورت عادی در فرمت جدولی چاپ کنند.

۷–۱۰ با استفاده از یک آرایه یک بعدی مسئله زیر را حل کنید: یک شرکت به فروشنده‌گان خود برحسب کمیسیون مبالغی پرداخت می‌کند. هر فروشنده برای هر هفته 200 دلار به همراه 9 درصد از فروش خود در آن هفته دریافت می‌کند. برای مثال اگر فروشنده‌ای در یک هفته 5000 دلار فروش داشته باشد، مبلغ 200 دلار به همراه 9 درصد از 500 دلار یعنی 650 دلار یافت خواهد کرد. برنامه‌ای بنویسید (با استفاده از یک آرایه از شمارنده‌ها) که تعداد فروشنده‌گان را برحسب محدوده‌های مشخص شده تعیین نماید:

a) $200 - $299
b) $300 - $399
c) $400 - $499
d) $500 - $599
e) $600 - $699
f) $700 - $799
g) $800 - $899
h) $900 - $999
i) $1000 و بالاتر.

۷–۱۱ در الگوریتم مرتب‌سازی حبابی (bubble sort)، مقادیرکوچکتر همانند حباب خود را به بالای آرایه می‌رسانند، همانند حرکت حباب در درون آب، در حالیکه مقادیر بزرگتر به پایین آرایه فرستاده می‌شوند. در مرتب‌سازی حبابی، چندین بار کل آرایه پیمایش می‌شود. در هر بار، جفت عناصر با هم مقایسه می‌شوند. اگر جفتی در ترتیب صعودی قرار داشته باشند (یا مقادیر یکسان باشند)، مقادیر در سر جای خود رها می‌شوند. اگر جفتی در ترتیب نزولی قرار داشته باشند، این مقادیر جای خود را در آرایه عوض می‌کنند. برنامه‌ای بنویسید که آرایه‌ای با 10 مقدار صحیح، را به روش حبابی، مرتب کند.

۷–۱۲ برنامه مرتب‌سازی حبابی ارائه شده، در آرایه‌های بزرگ فاقد کارائی لازم است. با بکار بردن اصلاحات ساده ارائه شده در زیر کارائی این نوع از مرتب‌سازی را افزایش دهید:

a) پس از اولین ارسال (گذار)، مطمئن هستیم بزرگترین عدد در بالاترین محل آرایه قرار دارد. پس از دومین گذار دو عدد بزرگ در آن مکان قرار دارند. و به همین ترتیب بجای انجام 9 مقایسه در هر بار گذار، مرتب‌سازی را طوری اصلاح کنید که در دومین گذار هشت مقایسه، در سومین گذار هفت مقایسه صورت گیرد و اینکار تا انتها انجام شود.

b) داده‌های موجود در آرایه ممکن است بصورت مرتب قرار گرفته باشند یا تقریباً دارای حالت مرتب شده باشند. پس چرا باید 9 گذار انجام گیرد که کمترین تأثیر را بر روی مرتب‌سازی اعمال می‌کند. مرتب‌سازی حبابی را به نحوی اصلاح کنید تا در پایان هر گذار تست کند که آیا عمل جابجایی صورت گرفته است یا خیر. اگر جابجایی صورت نگرفته باشد، پس داده‌ها در آرایه بصورت مرتب قرار گرفته‌اند پس برنامه باید خاتمه پذیرد. اگر جابجایی صورت گرفته باشد نیاز به یک گذار یا بیشتر داریم.

۷-۱۳ عباراتی بنویسید که موارد خواسته شده در زیر را بر روی یک آرایه تک بعدی اعمال کنند:

a) مقداردهی اولیه 10 عنصر آرایه **counts** با صفر.

b) افزودن 1 به هر 15 عنصر آرایه **bonus**.

c) خواندن 12 مقدار برای آرایه **monthlyTemperatures** از نوع **double** و از طریق صفحه‌کلید.

d) چاپ 5 مقدار از آرایه صحیح بنام **bestScores** در فرمت جدولی.

۷-۱۴ خطا یا خطاهای موجود در عبارات زیر را پیدا کنید:

a) با فرض اینکه: char str[5];

```
cin>>str; //user types "hello"
```

b) با فرض اینکه: int a[3];

```
cout<<a[1]<<" "<<a[2]<<" "<<a[3]<<endl;
```

```
double f[3]={1.1,10.01,100,001,1000,0001};
```

c)

d) با فرض اینکه: double d[2][10];

```
d[1,9]=2.345;
```

۷-۱۵ با استفاده از یک آرایه یک بعدی مسئله زیر را حل کنید: برنامه 20 عدد دریافت می‌کند که هر عدد باید مابین 100 و 10 باشد. اگر هر عدد دریافتی، تکرار نشده باشد، آنرا به نمایش درآورد. سعی کنید از کوچکترین آرایه استفاده کنید.

۷-۱۷ برنامه‌ای بنویسید که پرتاب دو طاس را شبیه‌سازی کند. در این برنامه باید از تابع **rand** در پرتاب طاس اول و مجدداً در پرتاب طاس دوم استفاده شود. سپس مجموع دو مقدار محاسبه شود. [نکته: هر طاس می‌تواند یک مقدار صحیح از 1 تا 6 را نشان دهد، از اینرو مجموع دو مقدار مابین 2 تا 12 متغیر خواهد بود] در شکل ۳۲-۷، ترکیبی از 36 حالت ممکنه در پرتاب دو طاس نشان داده شده است. برنامه شما باید دو طاس را به تعداد 36000 بار پرتاب کند. از یک آرایه تک بعدی استفاده کنید تا تعداد دفعات ممکنه از مجموع هر پرتاب را نشان دهد. نتایج را در فرمت جدولی چاپ کنید.

شکل ۳۲-۷| ۳۶ حالت ممکنه در پرتاب دوطاس.

۷-۱۸ برنامه زیر چه کاری انجام می‌دهد؟

```
// Ex. 7.18: ex07_18.cpp
// What does this program do?
#include <iostream>
using std::cout;
using std::endl;

int whatIsThis( int [], int ); // function prototype
```

```
int main()
{
   const int arraySize = 10;
   int a[ arraySize ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

   int result = whatIsThis( a, arraySize );

   cout << "Result is " << result << endl;

   return 0; // indicates successful termination
} // end main

// What does this function do?
int whatIsThis( int b[], int size )
{
   if ( size == 1 ) // base case
      return b[ 0 ];
   else // recursive step
      return b[ size - 1 ] + whatIsThis( b, size - 1 );
} // end function whatIsThis
```

۷-۱۹ برنامه شکل ۱۱-۶ را برای 6000 بار بازی craps تغییر دهید.

۷-۲۰ خط هوایی کوچکی مبادرت به خرید یک کامپیوتر برای سیستم رزرواسیون اتوماتیک خود کرده است. از شما خواسته شده تا این سیستم جدید را برنامه‌نویسی کنید. برنامه‌ای خواهید نوشت که صندلی‌های هر پرواز را تخصیص دهد (ظرفیت: 10 صندلی).

برنامه شما باید منوهای زیر را در اختیار کاربر قرار دهد:

Please type 1 for "first class"

و

Please type 2 for "Economy"

اگر کاربر، 1 را تایپ کند، برنامه یک صندلی در بخش First class (صندلی های1-5) به وی تخصیص خواهد داد. اگر کاربر، 2 را تایپ کند، برنامه یک صندلی در بخش Economy (صندلی های6-10) به وی تخصیص خواهد داد. برنامه باید اطلاعات صندلی از جمله شماره صندلی و نوع کلاس آن را چاپ کند.

برای نمایش صندلی‌ها در هواپیما از یک آرایه تک بعدی استفاده کنید. تمام عناصر آرایه را در ابتدای کار با صفر مقداردهی اولیه کنید تا نشان داده شود که همه صندلی‌ها خالی هستند. همانطوری که هر صندلی تخصیص داده می‌شود، موقعیت آن صندلی در آرایه با 1 تنظیم شود تا مشخص گردد که دیگر این صندلی را نمی‌توان به دیگری تخصیص داد. زمانیکه بخش First class پر شد، برنامه باید از کاربر سوال کند که آیا مایل به پذیرش صندلی در بخش Economg است یا خیر (برعکس اینحالت را هم انجام دهد). اگر پاسخ مثبت باشد، صندلی مربوطه تخصیص داده شود و در غیر اینصورت پیغام "Next flight leaves in 3 hours" را چاپ کند.

۷-۲۱ برنامه زیر چه کاری انجام می‌دهد؟

```
// Ex. 7.21: ex07_21.cpp
// What does this program do?
#include <iostream>
using std::cout;
using std::endl;

void someFunction( int [], int, int ); // function prototype

int main()
{
```

```
   const int arraySize = 10;
   int a[ arraySize ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

   cout << "The values in the array are:" << endl;
   someFunction( a, 0, arraySize );
   cout << endl;

   return 0; // indicates successful termination
} // end main

// What does this function do?
void someFunction( int b[], int current, int size )
{
   if ( current < size )
   {
      someFunction( b, current + 1, size );
      cout << b[ current ] << "  ";
   } // end if
} // end function someFunction
```

۷-۲۲ با استفاده از یک آرایه دو بعدی، مسئله زیر را حل کنید. شرکتی دارای چهار فروشنده است (1 تا 4) که مسئول فروش پنج محصول مختلف هستند (1 تا 5) در پایان یک روز، هر فروشنده صورت فروش هر نوع محصول را ارائه می کند.

هر صورت حاوی اطلاعات زیر است:

a) شماره فروشنده

b) شماره محصول

c) مبلغ کل از آن محصول در روز

بنابراین، هر فروشنده صورت فروشی مابین 0 تا 5 را ارائه می‌کند. فرض کنید اطلاعات تمام صورت فروش‌های ماه قبل در دسترس است. برنامه‌ای بنویسید که تمام این اطلاعات را خوانده و کل فروش هر فروشنده و محصول فروخته شده را بطور خلاصه بنمایش در آورد. کل فروش باید در آرایه دو بعدی **sales** ذخیره شده باشد. پس از پردازش کل اطلاعات ماه قبل، نتایج در فرمت جدولی که هر ستون نشاندهنده یک فروشنده خاص است و هر سطر نشاندهنده یک محصول ویژه، چاپ گردد.

۷-۲۳ زبان logo که از محبوبیت خاصی در مدارس ابتدایی برخوردار است، از مفهوم لاک‌پشت گرافیکی استفاده کرده است. تصور کنید که یک لاک‌پشت مکانیکی در فضائی حرکت می‌کند که در کنترل یک برنامه c++ است. لاک‌پشت مدادی را به یکی از دو جهت، بالا یا پایین حرکت می‌دهد. زمانیکه مداد به طرف پایین کشیده می‌شود، لاک‌پشت اشکالی را ترسیم می‌کند. با حرکت مداد به سمت بالا، لاک‌پشت آزادانه بدون ترسیم چیزی حرکت می‌کند. در این مسئله، می‌خواهیم حرکت لاک‌پشت را شبیه‌سازی کرده و صفحه طراحی کامپیوتری شده‌ای را هم برای آن ایجاد کنید.

از آرایه 20 در20 بنام **floor** استفاده کنید که در ابتدا با صفر مقداردهی اولیه شده است. دستورات از آرایه‌ای خوانده شود که حاوی آنها است. همیشه مسیر و موقعیت جاری لاک‌پشت را خواه مداد بالا باشد یا پایین، داشته باشید. فرض کنید لاک‌پشت از موقعیت (0,0) با مداد بالا حرکت خود را آغاز می‌کند. مجموعه دستورات لاک‌پشت که باید برنامه مبادرت به پردازش آنها کند در جدول ۳۳-۷ آورده شده‌اند.

فرض کنید که لاک‌پشت جای در نزدیکی مرکز صفحه قرار دارد. برنامه زیر باید یک مربع 12 در12 ترسیم و چاپ کرده و با بالا رفتن مداد به کار پایان دهد:

```
2
5,12
3
5,12
3
5,12
3
5,12
1
6
9
```

همانطوری که لاک‌پشت با مداد پایین حرکت می‌کند، عناصر مقتضی آرایه **floor** با 1 تنظیم می‌شوند. زمانیکه دستور 6 اعمال می‌شود (چاپ)، هر جا که 1 در آرایه وجود داشته باشد، یک کاراکتر ستاره یا کاراکتر دلخواه بنمایش در آید. هر جا که صفر در آرایه وجود داشته باشد، یک جای خالی بنمایش در آید. برنامه‌ای بنویسید که لاک گرافیکی را با قابلیت‌های فوق پیاده‌سازی کند.

مفهوم دستور	دستور
مداد بالا	1
مداد پایین	2
گردش به راست	3
گردش به چپ	4
حرکت به میزان 10 space(یا هر عددی به جای 10)	5.10
چاپ آرایه 20 در 20	6
پایان داده (مراقبتی)	9

شکل ۳۳-۷ | دستورات لاک‌پشت گرافیکی.

۷-۲۴ یکی از معماهای جالب در صفحه شطرنج، مسئله حرکت مهره اسب است. مسئله این است: آیا می‌توان مهره اسب را در صفحه خالی شطرنج به نحوی حرکت داد که کل 64 خانه صفحه را فقط یکبار لمس کرده باشد؟ در این تمرین به بررسی این مسئله می‌پردازیم. مهره اسب حرکتی بفرم L دارد. از اینرو، از یک خانه در میانه صفحه خالی شطرنج، این مهره می‌تواند هشت حرکت مختلف انجام دهد که در شکل ۳۴-۷ نشان داده شده‌اند (شماره گذاری شده از صفر تا هفت).

شکل ۳۴-۷ |هشت حرکت مختلف اسب.

a) یک صفحه شطرنج 8 در 8 بر روی کاغذ ترسیم کرده و مبادرت به حرکت مهره اسب بر روی آن کنید. در اولین خانه که اسب در آن فرود آمده عدد 1، در دومین خانه 2، در سومین خانه عدد 3، و الی آخر، قرار دهید. قبل از شروع حرکت، میزان و تعداد حرکت را تخمین بزنید. بخاطر داشته باشید که حرکت کامل برابر 64 حرکت است. تا کجا توانسته‌اید پیش بروید؟ آیا به حدثی که زده‌اید، نزدیک هستید؟

b) اکنون اجازه دهید، تا برنامه‌ای ایجاد کنیم که مهره اسب را در صفحات شطرنج به حرکت در آورد. صفحه شطرنج توسط یک آرایه دو بعدی 8 در 8 بنام board عرضه می‌شود. هر کدامیک از خانه‌های شطرنج با صفر مقداردهی اولیه می‌شوند. هر هشت حرکت ممکنه را در جهات افقی و عمودی بیان می‌کنیم. برای مثال، حرکت از نوع صفر، که در شکل ۳۴-۷ نشان داده شده است. متشکل از حرکت دو خانه افقی به سمت راست و یک خانه عمودی به سمت بالا می‌باشد. حرکت 2 متشکل از حرکت یک خانه افقی به چپ و دو خانه عمودی به بالا می‌باشد. حرکت افقی به چپ و حرکت عمودی به بالا دلالت بر مقادیر منفی دارند. این هشت حرکت را می‌توان توسط دو آرایه یک بعدی، **vertical**, **horizontal** توصیف کرد، همانند:

```
horizontal[0] = 2
horizontal[1] = 1
horizontal[2] = -1
horizontal[3] = -2
horizontal[4] = -2
horizontal[5] = -1
horizontal[6] = 1
horizontal[7] = 2

vertical[0] = -1
vertical[1] = -2
vertical[2] = -2
vertical[3] = -1
vertical[4] = 1
vertical[5] = 2
vertical[6] = 2
vertical[7] = 1
```

متغیرهای **currentColumn**, **currentRow** می‌توانند نشاندهنده موقعیت جاری سطر و ستون اسب باشند. برای داشتن حرکتی از نوع **moveNumber** که در آن **moveNumber** عددی مابین 0 و 7 است، برنامه شما از عبارت زیر استفاده کند.

```
currentRow += vertical [moveNumber];
current Column += horizontal [moveNumber];
```

از یک شمارنده که از 1 تا 64 در حال شمارش است، استفاده کنید. آخرین شماره را که اسب در آن مربع فرود آمده است را ثبت نمائید. بخاطر داشته باشید که هر حرکتی را تست کنید تا متوجه شوید که آیا اسب قبلاً در آن خانه حضور داشته است یا خیر، همچنین تست کنید که اسب از صفحه شطرنج خارج نشود. اکنون برنامه‌ای بنویسید

که مهره اسب را در صفحه شطرنج به حرکت در آورد. برنامه را اجرا کنید. اسب چند حرکت انجام داده است؟

c) پس از نوشتن و اجرای برنامه حرکت مهره اسب، محتملاً دید ارزشمندی بدست آورده‌اید. ما از این بینش برای توسعه یک استراتژی یا روش غیرمستدل در حرکت مهره اسب، استفاده خواهیم کرد. استراتژی تضمین کننده موفقیت نیست، اما اگر این استراتژی بدقت توسعه یافته باشد، شانس موفقیت نیز افزایش خواهد یافت. شاید متوجه شده باشید که خانه‌های خارجی به نسبت خانه‌های نزدیک به مرکز صفحه شطرنج، پر دردسرتر هستند. در واقع، پر زحمت‌ترین یا غیر قابل دسترس‌ترین، خانه‌ها در چهار گوشه قرار دارند.

چنین بینشی به شما پیشنهاد می‌دهد که ابتدا مهره اسب را به خانه‌های پر دردسر حرکت داده و سپس به سراغ خانه‌های آسانتر بروید، از اینرو زمانیکه صفحه شطرنج در انتهای حرکات فشرده می‌شود، شانس زیادی برای موفقیت بدست می‌آید.

می‌توانید با طبقه‌بندی کردن هر خانه مطابق با نحوه دسترسی به آن و سپس حرکت اسب به خانه‌ای (البته به شکل L) که تقریباً غیر قابل دسترسی است یک " استراتژی دسترسی" طراحی کنید. از یک آرایه دو بعدی بنام accessibility با اعدادی استفاده می‌کنیم که دلالت بر تعداد دفعات دسترسی هر خانه را در عمل نشان می‌دهند. در یک صفحه خالی شطرنج، هر خانه مرکزی با 8، هر خانه گوشه با 2 و سایر خانه‌ها با اعداد دسترسی 3،4 یا 6 درجه‌بندی شده‌اند، همانند:

2	3	4	4	4	4	3	2
3	4	6	6	6	6	4	3
4	6	8	8	8	8	6	4
4	6	8	8	8	8	6	4
4	6	8	8	8	8	6	4
4	6	8	8	8	8	6	4
3	4	6	6	6	6	4	3
2	3	4	4	4	4	3	2

حال نسخه‌ای از برنامه حرکت مهره اسب را بنویسید که از استراتژی دسترسی در آن استفاده شده باشد.

هر زمان، مهره اسب باید به خانه‌ای با پایین‌ترین عدد دسترسی منتقل شود. بنابر این حرکت را می‌توان از هر یک از چهار گوشه صفحه آغاز کرد. این نسخه از برنامه را اجرا کنید. آیا حرکت کامل را انجام داده‌اید؟ اکنون برنامه را برای 64 حرکت از یکی از گوشه‌ها آغاز کنید.

d) نسخه‌ای از برنامه حرکت مهره اسب را بنویسید که در زمان مواجه شدن مابین دو یا چند خانه، تصمیم بگیرد که کدام خانه انتخاب شود، با توجه به خانه‌های که از آن محل در دسترس هستند.

۷-۲۵ در تمرین ۷-۲۴ راه حلی برای مسئله حرکت مهره اسب در شطرنج ارائه کردیم. از روش " استراتژی دسترسی" استفاده کردیم که از کارایی و قابلیت مناسبی برخوردار است.

همانطوری که هر روز بر قدرت کامپیوترها افزوده می‌شود، ما هم قادر می‌شویم تا مسائل پیچیده‌تر را با توجه به قدرت کامپیوتر و الگوریتم‌های نه چندان حرفه‌ای‌تر حل کنیم، که این روش "brute force" نامیده می‌شود.

a) با استفاده از تولید عدد تصادفی اقدام به حرکت دادن تصادفی مهره اسب بر روی صفحه شطرنج کنید. برنامه شما

باید یک تور انجام داده و صفحه پایانی را چاپ کند. مهره تا به کجا حرکت کرده است؟

b) به احتمال زیاد، برنامه قبلی تور کوتاهتری را تولید می‌کرد. اکنون برنامه خود را برای انجام 1000 تور تغییر دهید. از یک آرایه تک بعدی برای حفظ تعداد تورها در هر مسیر استفاده کنید. پس از اینکه برنامه مبادرت به انجام 1000 تور کرد، بایستی این اطلاعات را بصورت مرتب در یک فرمت جدولی چاپ کند. بهترین نتیجه کدام است؟

c) به احتمال زیاد، برنامه قبلی تورهای قابل قبولی عرضه می‌کند، اما این تورها کامل نیستند. اکنون تورهای ناقص را خارج کرده و به برنامه اجازه دهید تا تولید یک تور کامل به کار خود ادامه دهد. [هشدار: این نسخه از برنامه بر روی یک کامپیوتر قدرتمند در حدود چند ساعت زمان صرف خواهد کرد.]

d) نسخه "brute force" را با نسخه "استراتژی دسترسی" در حرکت مهره شطرنج مقایسه کنید. کدامیک را ترجیح می‌دهید؟ توسعه کدام الگوریتم مشکل‌تر است؟ کدامیک به توان کامپیوتر بیشتر احتیاج دارد؟

۷-۲۶ یکی دیگر از مسائل صفحه شطرنج، مسئله هشت ملکه است، به اینصورت: آیا امکان دارد که هشت ملکه را بر روی صفحه خالی شطرنج به نحوی قرار داد که هیچ ملکه‌ای به ملکه دیگر حمله نکند، یعنی هیچ دو ملکه در یک سطر، ستون یا در امتداد یک خط مورب قرار نگیرد؟ با استفاده از تفکر ارائه شده در تمرین ۷-۲۴ یک استراتژی برای حل مسئله هشت ملکه بدست آورید. برنامه اجرا کنید.

۷-۲۷ در این تمرین، می‌خواهیم که به روش brute force مسئله هشت ملکه که در تمرین ۷-۲۷ گفته شده بپردازید.

a) با استفاده از تکنیک تولید عدد تصادفی که در تمرین ۷-۲۵ بیان شده، به حل تمرین هشت ملکه بپردازید.

b) از یک تکنیک جامع استفاده کنید، یعنی تمام حالت ممکنه از ترکیب هشت ملکه بر روی صفحه شطرنج را در نظر بگیرید.

c) چرا گمان می‌کنید روش جامع نمی‌تواند برای حل مسئله حرکت مهره اسب مناسب باشد؟

d) روش های b,a را با هم مقایسه کنید.

۷-۲۸ در مسئله حرکت مهره اسب، یک تور کامل زمانی اتفاق می‌افتد که مهره اسب در هر 64 خانه صفحه شطرنج فقط یک بار فرود آمده باشد. حرکت closed tour زمانی رخ می‌دهد که شصت‌وچهارمین حرکت یک خانه از مکان شروع حرکت مهره اسب فاصله داشته باشد. برنامه حرکت مهره اسب نوشته شده در تمرین ۷-۲۴ را برای تست closed tour اصلاح کنید، اگر یک تور کامل رخ داده باشد.

۷-۲۹ عدد اول، عددی است که همواره برخودش و 1 قابل تقسیم باشد. روش sieve of Eratosthense روشی برای یافتن اعداد اول است. این روش بصورت زیر عمل می‌کند:

a) ایجاد یک آرایه که تمام عناصر آن با 1 مقداردهی اولیه شده‌اند(true). عناصر آرایه با شاخص عدد اول به همان صورت یعنی 1 نگهداری می‌شوند. دیگر عناصر آرایه سرانجام با صفر تنظیم خواهند شد. شما می‌توانید عناصر 0 و 1 را در این تمرین نادیده بگیرید.

b) کار با شاخص 2 شروع می‌شود، هر زمان که یک عنصر آرایه یافته شد که مقدار آن 1 است، حلقه در مابقی آرایه حرکت کرده و هر عنصری را که شاخص آن مضربی از شاخص برای عنصری با مقدار 1 است، با صفر تنظیم

می‌کند. در آرایه با شاخص 2، تمام عناصر که مضربی از 2 هستند. با صفر تنظیم می‌شوند (شاخص‌های 4, 6, 8, 10 و الی آخر)، در آرایه با شاخص 3، تمام عناصر که مضربی از 3 هستند با صفر تنظیم می‌شوند (شاخص‌های 6, 9, 12, 15 و الی آخر) و همینطور تا آخر.

زمانیکه این فرآیند کامل شد، عناصر آرایه که هنوز 1 باقی مانده‌اند نشان می‌دهند که شاخص یک عدد اول است. می‌توان این شاخص‌ها را چاپ کرد. برنامه‌ای بنویسید که از یک آرایه 100 عنصری برای تعیین و چاپ اعداد اول مابین 2 تا 999 استفاده کند. عنصر صفر آرایه را در نظر نگیرید.

۷-۳۰ مرتب‌سازی باکت (bucket sort) بر روی یک آرایه تک بعدی از مقادیر مثبت صحیح شروع و آنرا مرتب می‌کند و بر روی یک آرایه دو بعدی از مقادیر صحیح با سطرهای شاخص‌گذاری شده از 0 تا 9 و ستون‌های شاخص‌گذاری شده از صفر تا n-1 شروع می‌شود که n تعداد مقادیر در آرایه است که مرتب خواهد شد. به هر سطر آرایه دو بعدی، یک باکت (bucket) گفته می‌شود. تابعی بنام **bucketSort** بنویسید که یک آرایه صحیح و سایز آرایه را بعنوان آرگومان دریافت کرده و مراحل زیر را انجام دهد:

a) مقدار هر آرایه یک بعدی را در سطری از باکت آرایه و بر مبنای رقم یکان مقدار قرار دهد. برای مثال، 97 در سطر 7، 3 در سطر 3 و 100 در سطر صفر جای داده شود. به اینحالت "گذر توزیعی" گفته می‌شود.

b) سطر به سطر از میان باکت آرایه عبور کرده و مقادیر را به آرایه اصلی کپی کنید به اینحالت "گذر تجمعی" گفته می‌شود. ترتیب جدید مقادیر فوق الذکر یک بعدی بصورت 100, 3, 97 خواهد بود.

c) این فرآیند را برای هر موقعیت مکانی رقم تکرار کنید (دهگان، صدگان، هزارگان، ...)

در دومین گذر، 100 در سطر صفر، 3 در سطر صفر (چرا که 3 دارای رقم دهگان نیست) و 97 در سطر 9 جای داده می‌شود. پس از گذر تجمعی، ترتیب مقادیر در آرایه یک بعدی بصورت 100, 3, 97 خواهد بود. در سومین گذر، 100 در سطر 1، 3 در سطر صفر و 97 در سطر صفر جای خواهد گرفت (پس از 3). پس از آخرین گذر تجمعی، آرایه اصلی مرتب شده خواهد بود و تکنیک بکار رفته در روش باکت به نسبت مرتب‌سازی درجی از کارایی بهتری برخوردار است، اما نیازمند حافظه بیشتر می‌باشد.

تمرینات بازگشتی

۷-۳۱ در مرتب‌سازی انتخابی (selection sort) آرایه بدنبال کوچکترین عنصر جستجو می‌شود. سپس جای کوچکترین عنصر با اولین عنصر در آرایه عوض می‌شود. این فرآیند برای زیرآرایه که با دومین عنصر آرایه آغاز می‌شود، تکرار می‌گردد. در هر گذر آرایه یک عنصر در مکان صحیح خود قرار داده می‌شود. زماینکه زیرآرایه به یک عنصر ختم شود، پس آرایه مرتب شده است. تابع بازگشتی **selectionSort** را برای انجام این الگوریتم بنویسید.

۷-۳۲ پالندروم رشته‌ای است که تلفظ آن از ابتدا و هم از انتها یکسان است. برای مثال عبارات زیر همگی پالندروم هستند: "radar"، "able was i ere i saw elba" و اگر فاصله‌ها را نادیده بگیریم عبارت "a man a plan a canal panama". تابع بازگشتی بنام testPalindrome بنویسید که اگر رشته ذخیره شده در آرایه، پالندروم باشد، مقدار true بازگرداند و در غیر اینصورت مقدار false. تابع باید فضاهای خالی را در نظر نگیرد.

۷-۳۳ (جستجوی خطی) برنامه ۷-۱۹ را با استفاده از تابع بازگشتی **linearSearch** اصلاح کنید. این تابع باید یک آرایه از نوع صحیح، یک کلید جستجو، شاخص آغازین و شاخص پایانی را به عنوان آرگومان دریافت کند. اگر کلید جستجو یافت شود، شاخص آرایه بعنوان پاسخ برگشت داده شود و در غیر اینصورت مقدار 1- چاپ گردد.

۷-۳۴ برنامه هشت ملکه ایجاد شده در تمرین ۷-۲۶ را بصورت بازگشتی پیاده‌سازی کنید.

۷-۳۵ یک تابع بازگشتی بنام **printArray** بنویسید که یک آرایه، شاخص شروع و شاخص پایانی را بعنوان آرگومان دریافت کرده و چیزی برگشت ندهد. تابع باید زمانی پردازش را متوقف و برگشت یابد که شاخص شروع معادل با شاخص پایانی باشد.

۷-۳۶ تابع بازگشتی بنام **stringReverse** بنویسید که یک آرایه کاراکتری حاوی یک رشته و شاخص شروع را به عنوان آرگومان دریافت کرده، رشته را بصورت معکوس چاپ کرده و چیزی برگشت ندهد. تابع باید زمانی به پردازش خاتمه دهد که با کاراکتر null مواجه شده باشد.

۷-۳۷ تابع بازگشتی بنام **recursiveMinimum** بنویسید که یک آرایه صحیح، شاخص شروع و پایان را بعنوان آرگومان دریافت و کوچکترین عنصر آرایه را برگشت دهد. تابع باید زمانی به پردازش خاتمه دهد که شاخص شروع معادل با شاخص پایان باشد.

تمرینات vector

۷-۳۸ از یک **vector** صحیح برای حل مسئله توضیح داده شده در تمرین ۷-۱۰ استفاده کنید.

۷-۳۹ برنامه پرتاب طاس مطرح شده در تمرین ۷-۱۷ را برای استفاده از **vector** به منظور ذخیره‌سازی تعداد دفعات مجموع پرتاب طاس‌ها اصلاح کنید.

۷-۴۰ راه حل مطرح شده در تمرین ۷-۳۷ را برای یافتن کوچکترین مقدار در یک **vector** بجای یک آرایه اصلاح کنید.

فصل هشتم

اشاره‌گرها و رشته‌های مبتنی بر اشاره‌گر

اهداف

- اشاره‌گرها چیستند.
- شباهت و تفاوت مابین اشاره‌گرها و مراجعه‌ها و زمان استفاده از آنها.
- کاربرد اشاره‌گرها برای ارسال آرگومان‌ها به توابع به روش مراجعه.
- استفاده از رشته‌های مبتنی بر اشاره‌گر.
- رابطه نزدیک مابین اشاره‌گرها، آرایه‌ها و رشته‌ها.
- کاربرد اشاره‌گرها در توابع.
- اعلان و استفاده از آرایه‌های رشته‌ای.

رئوس مطالب

۸-۱ مقدمه

بحث این فصل در مورد یکی از مهمترین و قویترین ویژگیهای زبان برنامه نویسی C++، یعنی *اشاره‌گرها* (*pointer*) است. در فصل ششم مشاهده کردید که مراجعه‌ها می‌توانند برای انجام عملیات ارسال به روش *مراجعه* بکار گرفته شوند. اشاره‌گرها هم می‌توانند چنین عملیاتی انجام دهند و می‌توانند برای ایجاد و دستکاری ساختمان داده‌های دینامیکی (پویا) همانند لیست پیوندی، صف‌ها، پشته‌ها و درخت‌ها بکار گرفته شوند. در این فصل به توضیح مفاهیم پایه و بررسی رابطه موجود مابین آرایه‌ها و اشاره‌گرها خواهیم پرداخت. بررسی آرایه‌ها بعنوان اشاره‌گر از زبان برنامه‌نویسی C مشتق شده است. همانطوری‌که در فصل هفتم مشاهده کردید، کلاس **vector** مبادرت به پیاده‌سازی آرایه‌ها به روش خاصی می‌کند. به همین ترتیب، C++ دو نوع رشته عرضه می‌کند، شی‌های از کلاس **string** (که در فصل سوم از آنها استفاده کردیم) و رشته‌های مبتنی بر اشاره‌گر **char *** به سبک C. بحث این فصل بر روی رشته‌های اشاره‌گر **char *** متمرکز است تا دانش شما در ارتباط با اشاره‌گرها عمیق‌تر شود. در واقع، رشته‌های مطرح شده در بخش ۴-۷ و بکار رفته در برنامه ۱۲-۷ از نوع رشته‌های مبتنی بر اشاره‌گر **char *** بودند.

در فصل سیزدهم به استفاده از اشاره‌گرها در کلاس‌ها خواهیم پرداخت که به آن در برنامه‌نویسی شی‌گرا «*پردازش چند ریختی*» می‌گویم. در فصل بیست و یکم مثالهای در ارتباط با ایجاد و استفاده از ساختارهای داده پویا که با اشاره‌گرها پیاده‌سازی می‌شوند مطرح کرده‌ایم.

۸-۲ اعلان و مقداردهی اولیه متغیرهای اشاره‌گر

متغیرهای اشاره‌گر حاوی آدرس‌های حافظه بعنوان مقادیر خود هستند. معمولاً، یک متغیر حاوی یک مقدار مشخص است. با این وجود، یک اشاره‌گر حاوی آدرس حافظه از متغیری است که دارای یک مقدار مشخص می‌باشد. در چنین وضعیتی، نام متغیر بصورت مستقیم به یک مقدار مراجعه دارد و یک اشاره‌گر بصورت غیرمستقیم به یک مقدار مراجعه می‌کند (شکل ۱-۸). غالباً مراجعه به یک مقدار از طریق یک اشاره‌گر بصورت غیرمستقیم انجام می‌شود. توجه کنید که در تصاویر، اشاره‌گر بصورت یک فلش از متغیری که حاوی یک آدرس به سمت متغیری که در آن آدرس حافظه قرار دارد نشان داده می‌شود. اشاره‌گرها، همانند هر متغیر دیگری، بایستی قبل از استفاده اعلان شده باشند. برای مثال، برای اشاره‌گر به نمایش درآمده در شکل ۱-۸ اعلان

```
int *countPtr, count;
```

متغیر **countPtr** را از نوع ***int** (یعنی یک اشاره‌گر به یک مقدار **int**) اعلان کرده که به اینصورت خوانده می‌شود «**countPtr** اشاره‌گری به **int** است» یا «**countPtr** اشاره‌گر به یک شی از نوع **int** است.» همچنین در این اعلان، متغیر **count** از نوع **int** اعلان شده است و اشاره‌گری به یک **int** نمی‌باشد. علامت * فقط بر روی **countPtr** اعمال شده است. هر متغیری که می‌خواهد بصورت یک اشاره‌گر اعلان شود باید قبل از آن یک علامت * قرار گرفته باشد. برای مثال، در اعلان

```
double *xPtr, *yPtr;
```

مشخص است که **xPtr** و **yPtr** هر دو اشاره‌گرهای به مقادیری از نوع **double** هستند. زمانیکه * در اعلانی ظاهر می‌شود، دیگر بعنوان یک عملگر مطرح نشده بلکه نشاندهنده متغیری است که بصورت یک اشاره‌گر اعلان شده است. اشاره‌گرها می‌توانند برای اشاره به شی‌های از هر نوع داده اعلان شوند.

خطای برنامه‌نویسی

*فرض اینکه استفاده از یک * در اعلان یک اشاره‌گر بر روی تمام اسامی متغیرها در یک اعلان جدا شده با کاما از یکدیگر تاثیر خواهد گذاشت، تصور اشتباهی است. هر اشاره‌گر باید با یک پیشوند *قبل از نام اعلان شده باشد (خواه با فاصله یا بدون فاصله، کامپایلر فاصله‌ها را نادیده می‌گیرد). اعلان یک متغیر در هر خط کمک می‌کند تا جلوی رخ دادن چنین اشتباهاتی گرفته شود و خوانایی برنامه افزایش یابد.*

برنامه‌نویسی ایده‌ال

*اگر چه انجام این کار ضروری نیست، اما بهتر است حروف **Ptr** را برای نشان دادن اینکه متغیر از نوع اشاره‌گر است با نام متغیر همراه کنید.*

شکل ۱-۸| مراجعه مستقیم و غیرمستقیم به یک متغیر.

بایستی اشاره‌گرها به هنگام اعلان یا توسط یک عبارت تخصیص‌دهنده مقداردهی اولیه شوند. یک اشاره‌گر می‌تواند با صفر، **NULL** یا یک آدرس مقداردهی اولیه شود. یک اشاره‌گر با مقدار صفر یا NULL به چیزی اشاره نمی‌کند و بعنوان یک *اشاره‌گر null* شناخته می‌شود. ثابت سمبولیک NULL در فایل سرآیند <**iostream**> و چند فایل سرآیند از کتابخانه استاندارد تعریف شده که نشاندهنده مقدار صفر است. مقداردهی اولیه یک اشاره‌گر با NULL معادل با مقداردهی اولیه یک اشاره‌گر با صفر است، اما در C++ بطور قراردادی از صفر استفاده می‌شود. زمانیکه صفر تخصیص داده می‌شود، به یک اشاره‌گر از نوع مقتضی برگردانده می‌شود. مقدار صفر تنها مقدار صحیحی است که می‌تواند مستقیماً به یک متغیر اشاره‌گر تخصیص داده شود بدون اینکه ابتدا نوع صحیح به نوع اشاره‌گر تبدیل شود.

اجتناب از خطا

برای اجتناب از اشاره اشاره‌گرها به مکان‌های ناشناخته و مقداردهی نشده حافظه، مبادرت به مقداردهی اولیه اشاره‌گرها کنید.

۳ـ ۸ عملگرهای اشاره‌گر

عملگر آدرس (**&**) یک عملگر غیربایتری است که آدرس حافظه علموند خود را برگشت می‌دهد. برای مثال، با فرض اعلان‌های

```
int y = 5; //declare variable y
int *yPtr; // declare pointer variable yPtr
```

عبارت

```
yPtr = &y; // assign address of y to yPtr
```

آدرس متغیر **y** را به متغیر اشاره‌گر **yPtr** تخصیص می‌دهد. پس متغیر **yPtr** به **y** اشاره می‌کند. اکنون، **yPtr** بصورت غیرمستقیم به مقدار متغیر **y** مراجعه دارد. به نحوه استفاده از **&** در عبارت انتساب فوق دقت کنید که مشابه کاربرد **&** در اعلان متغیر مراجعه‌ای نیست.

شکل ۲-۸ نمایشی از حافظه پس از تخصیص فوق است. رابطه اشاره‌گر توسط یک فلش یا پیکان از جعبه‌ای که نشاندهنده اشاره‌گر **yPtr** در حافظه به جعبه‌ای که نشاندهنده متغیر **y** در حافظه است، عرضه شده است.

شکل ۳-۸ نمایش دیگری از اشاره‌گر در حافظه است، با فرض اینکه متغیر صحیح **y** در مکان 600000 حافظه و متغیر اشاره‌گر **yPtr** در مکان 500000 حافظه ذخیره شده است. عملوند، عملگر آدرس بایستی یک *lvalue* (یعنی چیزی که بتوان مقداری به آن تخصیص داد همانند نام یک متغیر یا مراجعه) باشد، عملگر آدرس نمی‌تواند با ثابت‌ها یا عباراتی که نمی‌توانند در مراجعه نتیجه‌ای بدست دهند بکار گرفته شود.

شکل ۲-۸ | نمایش گرافیکی از اشاره یک اشاره‌گر به یک متغیر در حافظه.

شکل ۳-۸ | نمایش y و yPtr در حافظه.

معمولاً از عملگر * بعنوان یک عملگر غیرمستقیم یا عملگر غیرمراجعه‌ای یاد می‌شد که یک مترداف برای شی که عملوند اشاره‌گر به آن اشاره می‌کند، برگشت می‌دهد. برای مثال (به شکل ۲-۸ توجه کنید) عبارت

```
cout << *yPtr << endl;
```

مقدار متغیر y، یعنی 5 را همانند عبارت زیر چاپ می‌کند

```
cout << y << endl;
```

به استفاده از عملگر * به این روش، مراجعه غیرمستقیم اشاره‌گر گفته می‌شود. توجه کنید که مراجعه غیرمستقیم اشاره‌گر می‌تواند در سمت چپ یک عبارت تخصیصی بکار گرفته شود، همانند

```
*yPtr = 9;
```

که مقدار 9 را به y در شکل ۳-۸ تخصیص می‌دهد. همچنین می‌توان از این عملگر برای بازیابی مقدار ورودی استفاده کرد، برای مثال

```
cin >> *yPtr;
```

مقدار ورودی را در y قرار می‌دهد. اشاره‌گر غیرمراجعه‌ای یک *lvalue* است.

در برنامه شکل ۴-۸ به توضیح عملگرهای اشاره‌گر & و * پرداخته شده است. مکان‌های حافظه چاپ شده توسط >> در این مثال بصورت مقادیر هگزادسیمال (پایه 16) هستند. توجه کنید که آدرس‌های حافظه هگزادسیمال بدست آمده در این برنامه وابسته به کامپایلر و سیستم عامل هستند و امکان دارد برنامه شما نتایج متفاوتی از این برنامه تولید کند.

```
1  // Fig. 8.4: fig08_04.cpp
2  // Using the & and * operators.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     int a; // a is an integer
10    int *aPtr; // aPtr is an int * -- pointer to an integer
11
12    a = 7; // assigned 7 to a
13    aPtr = &a; // assign the address of a to aPtr
14
15    cout << "The address of a is " << &a
16       << "\nThe value of aPtr is " << aPtr;
17    cout << "\n\nThe value of a is " << a
18       << "\nThe value of *aPtr is " << *aPtr;
19    cout << "\n\nShowing that * and & are inverses of "
20       << "each other.\n&*aPtr = " << &*aPtr
21       << "\n*&aPtr = " << *&aPtr << endl;
22    return 0; // indicates successful termination
23 } // end main
```

```
The address of a is 0012F580
The value of aPtr is 0012F580

The value of a is 7
The value of *aPtr is 7
```

```
Showing that * and & are inverses of each other.
&*aPtr = 0012F580
*&aPtr = 0012F580
```

شکل ٤-٨| عملگرهای & و *.

دقت کنید که آدرس **a** (خط 15) و مقدار **aPtr** (خط 16) در خروجی یکسان هستند و تایید می‌کنند که آدرس **a** براستی به متغیر اشاره‌گر **aPtr** تخصیص یافته است. عملگرهای **&** و * وارون همدیگر هستند. زمانیکه هر دو آنها بصورت متوالی بر روی **aPtr** بکار گرفته شوند به هر ترتیب، سبب «لغو دیگری شده» و مقدار یکسانی (مقدار **aPtr**) چاپ می‌شود.

در جدول شکل ۵-۸ لیستی از تقدم و شرکت پذیری عملگرهای مطرح شده تا بدین جا آورده شده است. توجه کنید که عملگر آدرس (**&**) و عملگر غیرمراجعه‌ای (*) از جمله عملگرهای غیربایزی در سومین سطح از جدول تقدم عملگرها هستند.

عملگر	ارتباط	نوع
( )	**چپ به راست**	**پرانتز**
++ -- static_cast< *type* >()	**چپ به راست**	**غیرباینری**
++ -- + -	**راست به چپ**	**غیرباینری**
*** / %**	**چپ به راست**	**تعددی**
+ -	**چپ به راست**	**افزاینده کاهنده**
<< >>	**چپ به راست**	**درج/استخراج**
< <= > >=	**چپ به راست**	**رابطه‌ای**
== !=	**چپ به راست**	**برابری**
&&	**چپ به راست**	**ANDمنطقی**
||	**چپ به راست**	**ORمنطقی**
?:	**راست به چپ**	**شرطی**
= += -= *= /= %=	**راست به چپ**	**تخصیصی**
,	**چپ به راست**	**کاما**

شکل ۵-۸| تقدم و شرکت پذیری عملگرها.

٤-٨ ارسال آرگومان به توابع به روش مراجعه با اشاره‌گرها

در C++ سه روش برای ارسال آرگومان به یک تابع وجود دارد، ارسال با مقدار، *ارسال با مراجعه* با *آرگومان‌های مراجعه‌ای* و *ارسال با مراجعه* با *آرگومان‌های اشاره‌گر*.

در فصل ششم به مقایسه ارسال آرگومان مراجعه‌ای به روش مقدار و مراجعه پرداختیم. در این بخش به توضیح ارسال آرگومان‌های اشاره‌گر به روش مراجعه می‌پردازیم. همانطوری که در فصل ششم مشاهده کردید، **return** می‌توانست برای بازگرداندن یک مقدار از تابع فراخوانی شده به فراخوان (یا بازگرداندن

کنترل از تابع فراخوانی شده بدون برگشت دادن مقداری) بکار گرفته شود. همچنین شاهد بودید که آرگومان‌ها می‌توانستند به تابع با استفاده از آرگومان‌های مراجعه‌ای ارسال شوند. چنین آرگومان‌های به تابع فراخوانی شده امکان تغییر در مقادیر اصلی آرگومان‌ها در فراخوان را می‌دهند. همچنین آرگومان‌های مراجعه‌ای به برنامه‌ها امکان ارسال شی‌ها با داده‌های بزرگ به یک تابع را فراهم می‌آورند و از سربارگذاری ارسال شی به روش مقدار اجتناب می‌شود (که مستلزم ایجاد یک کپی از شی است). از اشاره‌گرها همانند مراجعه‌ها، می‌توان برای اصلاح یک یا چندین متغیر در فراخوان یا ارسال اشاره‌گرها به شی‌های با داده‌های بزرگ استفاده کرد تا از سربارگذاری ارسال شی‌ها به روش مقدار جلوگیری شود.

در C++، برنامه‌نویسان می‌توانند از اشاره‌گرها و عملگر غیرمستقیم (*) برای انجام ارسال به روش مراجعه استفاده کنند. در زمان فراخوانی تابعی با آرگومانی که باید اصلاح شود، آدرس آرگومان ارسال می‌شود. معمولاً اینکار با اعمال عملگر آدرس (&) بر روی نام متغیری که مقدار آن تغییر خواهد یافت صورت می‌گیرد.

همانطوری که در فصل هفتم مشاهده کردید، آرایه‌ها با استفاده از عملگر & قادر به ارسال نیستند، چرا که نام آرایه مکان شروع در حافظه برای آن آرایه است (یعنی نام آرایه خود یک اشاره‌گر است). نام یک آرایه همانند **arrayName**، معادل با **&arrayName[0]** است. زمانیکه آدرس یک متغیر به یک تابع ارسال می‌شود، عملگر غیرمستقیم (*) می‌تواند در تابع بفرم مترادفی از نام متغیر بکار گرفته شود، در اینحالت می‌تواند مقدار متغیر در آن مکان از حافظه فراخوان را تغییر دهد.

برنامه‌های شکل ۶-۸ و ۷-۸ دو نسخه از یک تابع را عرضه می‌کنند که مکعب دو مقدار صحیح **cubeByReference** و **cubeByValue** را محاسبه می‌کنند. در برنامه شکل ۶-۸ متغیر **number** به روش مقدار به تابع **cubeByValue** (خط 15) ارسال می‌شود. تابع **cubeByValue** (خطوط 24-21) مکعب آرگومان خود را محاسبه مقدار جدید را به **main** و با استفاده از دستور **return** برگشت می‌دهد (خط 23). مقدار جدید به **nubmer** در **main** تخصیص می‌یابد (خط 15). توجه کنید که تابع فراخوان فرصت بررسی نتیجه از فراخوان تابع را قبل از اصلاح مقدار متغیر **number** را دارد. برای مثال، در این برنامه، می‌توانیم نتیجه **cubeByValue** را در متغیر دیگری ذخیره کرده و به بررسی مقدار آن پرداخته و فقط در صورتیکه تشخیص دهیم که مقدار برگشتی قابل قبول است آنرا به **number** تخصیص دهیم.

```
1  Fig. 8.6: fig08_06.cpp
2  // Cube a variable using pass-by-value.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int cubeByValue( int ); // prototype
8
9  int main()
10 {
```

```
11    int number = 5;
12
13    cout << "The original value of number is " << number;
14
15    number = cubeByValue( number ); // pass number by value to cubeByValue
16    cout << "\nThe new value of number is " << number << endl;
17    return 0; // indicates successful termination
18 } // end main
19
20 // calculate and return cube of integer argument
21 int cubeByValue( int n )
22 {
23    return n * n * n; // cube local variable n and return result
24 } // end function cubeByValue
```

```
The original value of number is 5
The new value of number is 125
```

شکل ٦-٨| ارسال به روش مقدار برای محاسبه مکعب مقدار یک متغیر.

در برنامه شکل ۷-۸ متغیر **number** به تابع **cubeByReference** با استفاده از روش مراجعه با یک آرگومان اشاره‌گر ارسال شده است (خط 15). آدرس **number** به تابع ارسال می‌شود. تابع **cubeByReference** در خطوط 25-22 تصریح کننده پارامتر **nPtr** (یک اشاره‌گر به **int**) برای دریافت آرگومان خود است. تابع، اشاره‌گر را دنبال و مقدار مکعب که **nPtr** به آن اشاره می‌کند را محاسبه می‌نماید (خط 24). در اینحالت مقدار **nubmer** در **main** مستقیماً تغییر می‌یابد.

تابع دریافت کننده آدرس بعنوان یک آرگومان بایستی یک پارامتر اشاره‌گر به آدرس دریافتی تعریف کرده باشد. برای مثال در سرآیند تابع **cubeByReference** در خط 12 مشخص شده است که **cubeByReference** آدرس یک متغیر **int** (یعنی یک اشاره‌گر به یک **int**) را بعنوان آرگومان دریافت می‌کند، آدرس را بصورت محلی در **nPtr** ذخیره کرده و مقداری برگشت نمی‌دهد.

نمونه اولیه تابع برای **cubeByReference** در خط 7 حاوی **int *** در درون پرانتز است. همانند سایر انواع متغیرها، نیازی به شامل کردن اسامی پارامترهای اشاره‌گر در نمونه اولیه تابع نیست. هدف از قرار دادن اسامی فقط تهیه مستندات بوده و توسط کامپایلر نادیده گرفته می‌شوند. تصاویر ۸-۸ و ۹-۸ یک تحلیل گرافیکی به ترتیب از اجرای برنامه‌های ۶-۸ و ۷-۸ ارائه می‌کنند.

```
1  // Fig. 8.7: fig08_07.cpp
2  // Cube a variable using pass-by-reference with a pointer argument.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  void cubeByReference( int * ); // prototype
8
9  int main()
10 {
11    int number = 5;
12
13    cout << "The original value of number is " << number;
14
15    cubeByReference( &number ); // pass number address to cubeByReference
16
17    cout << "\nThe new value of number is " << number << endl;
18    return 0; // indicates successful termination
19 } // end main
20
```

```
21 // calculate cube of *nPtr; modifies variable number in main
22 void cubeByReference( int *nPtr )
23 {
24    *nPtr = *nPtr * *nPtr * *nPtr; // cube *nPtr
25 } // end function cubeByReference
```

```
The original value of number is 5
The new value of number is 125
```

شکل ۷-۸ | ارسال به روش مراجعه با یک آرگومان اشاره‌گر برای محاسبه مکعب یک متغیر.

شکل ۸-۸ | تحلیل ارسال به روش مقدار در برنامه شکل ٦-۸.

شکل ۹-۸ | تحلیل ارسال به روش مراجعه (با یک آرگومان اشاره‌گر) در برنامه شکل ۷-۸.

۸-۵ کاربرد const همراه با اشاره‌گرها

بخاطر دارید که توصیف‌کننده **const** به برنامه‌نویس امکان می‌دهد تا به کامپایلر اطلاع دهد که مقدار یک متغیر خاص نبایستی تغییر یابد.

قابلیت حمل

*اگرچه **const** بخوبی در **ANSI C** و **C++** تعریف شده است، اما برخی از کامپایلر آنرا بطور کامل رعایت نمی‌کنند. بهتر است به بررسی کامپایلر خود بپردازید.*

سالها بخش اعظمی از کدهای نوشته شده و به یادگار مانده از نسخه‌های اولیه C از وجود **const** تهی بودند چرا که وجود نداشت. به همین دلیل مجال مناسبی برای بهبود کد قدیمی C بوجود آمده است. همچنین هنوز هم برخی از برنامه‌نویسان استفاده کننده از ANSI C و C++ از **const** در برنامه‌های خود استفاده نمی‌کنند، چراکه آنها برنامه‌نویسی را با نسخه‌های اولیه C شروع کرده و یاد گرفته‌اند. چنین برنامه‌نویسانی از فرصت‌های مناسبی که یک مهندسی نرم‌افزار ایده‌ال می‌تواند در اختیار آنها قرار دهد، محروم هستند.

دلایل مختلفی برای استفاده (یا عدم استفاده) از **const** در کنار پارامترهای تابع وجود دارد. چگونه شرایط انتخاب یا عدم انتخاب را تشخیص می‌دهید؟ اجازه دهید تا به بررسی یک روش و قاعده علمی بپردازیم. همیشه برای یک تابع، آن میزان دسترسی به داده پارامترهای خود اعطاء کنید که بتواند وظیفه خود را به انجام برساند، نه بیشتر. بحث این بخش در ارتباط با نحوه ترکیب **const** با اعلان اشاره‌گر است تا حداقل قواعد علمی رعایت شود.

در فصل ششم توضیح داده شد که به هنگام فراخوانی یک تابع به روش ارسال با مقدار، یکی کپی از آرگومان (یا آرگومان‌ها) در فراخوان تابع ایجاد و به تابع ارسال می‌شود. اگر کپی در تابع تغییر یابد، مقدار اصلی همچنان در فراخوان بدون تغییر باقی می‌ماند. در برخی از موارد، یک مقدار ارسالی به تابع تغییر داده می‌شود تا اینکه تابع بتواند وظیفه خود را انجام دهد. با این همه، در برخی از شرایط نبایستی مقدار در تابع فراخوانی شده تغییر پیدا کند، حتی اگر تابع فراخوانی شده مبادرت به دستکاری کردن کپی از مقدار اصلی نماید.

برای مثال فرض کنید تابعی یک آرایه تک بعدی و سایز آنرا بعنوان آرگومان دریافت کرده و متعاقب آن آرایه را چاپ می‌کند. چنین تابعی باید با استفاده از یک حلقه در میان آرایه حرکت کرده و هر عنصر آرایه را بصورت مجزا در خروجی قرار دهد. از سایز در بدنه تابع برای تعیین بالاترین شاخص آرایه استفاده می‌شود، از اینروست که حلقه می‌تواند زمان کامل شدن عملیات چاپ را تشخیص دهد. سایز آرایه نمی‌تواند در بدنه تابع تغییر یابد، بنابر این بایستی بعنوان **const** اعلان شود. البته، بدلیل اینکه آرایه فقط چاپ می‌شود، بایستی خود آنرا را هم بصورت **const** اعلان کرده باشیم. اینکار از اهمیت خاصی برخوردار است چراکه کل آرایه همیشه بصورت مراجعه ارسال می‌شود و می‌تواند به آسانی در تابع فراخوانی شده تغییر داده شود.

مهندسی نرم‌افزار

*اگر مقداری در بدنه تابع که به آن ارسال شده تغییر نمی‌یابد (یا نبایستی تغییر داده شود)، باید آن پارامتر بصورت **const**اعلان شود تا از تغییر ناخواسته آن اجتناب گردد.*

اگر مبادرت به تغییر مقدار یک **const** شود، پیغام هشدار یا خطا با توجه به نوع کامپایلر صادر خواهد شد.

اجتناب از خطا

قبل از استفاده از یک تابع، به بررسی نمونه اولیه تابع بپردازید تا مشخص شود که پارامترها می‌توانند تغییر یابند یا خیر.

چهار روش برای ارسال یک اشاره‌گر به یک تابع وجود دارد: اشاره‌گر غیرثابت به داده غیرثابت (شکل ۱۰-۸)، اشاره‌گر غیرثابت به داده ثابت (شکل ۱۱-۸ و ۱۲-۸)، اشاره‌گر ثابت به داده غیرثابت (شکل ۱۳-۸) و اشاره‌گر ثابت به داده ثابت (شکل ۱۴-۸). هر یک از این حالت‌ها مجوزهای دسترسی متفاوتی بوجود می‌آورند.

اشاره‌گر غیرثابت به داده غیرثابت

بالاترین سطح دسترسی توسط یک اشاره‌گر غیرثابت به یک داده غیرثابت اهداء می‌شود. داده می‌تواند از طریق دسترسی به مقدار اشاره‌گر تغییر داده شود و اشاره‌گر می‌تواند برای اشاره به داده دیگری اصلاح یا تغییر داده شود. در اعلان یک اشاره‌گر غیرثابت به داده غیرثابت نیازی به حضور **const** نیست. چنین اشاره‌گری می‌تواند برای بازیابی یک رشته پایان یافته با **null** در یک تابع که مقدار اشاره‌گر را در ضمن پردازش هر کاراکتر در رشته تغییر می‌دهد، بکار گرفته شود. از بخش ۴-۷ بخاطر دارید که چنین رشته‌ای می‌تواند در یک آرایه کاراکتری که حاوی کاراکترهای آن رشته و یک کاراکتر **null** بعنوان نشاندهنده انتهای رشته است، جای داده شود.

در برنامه شکل ۱۰-۸، تابع **convertToUppercase** در خطوط 34-25 پارامتر **sPtr** (خط 25) را بصورت یک اشاره‌گر غیرثابت به یک داده غیرثابت اعلان کرده است (بدون حضور **const**). تابع مبادرت به پردازش یک کاراکتر در هر زمان از یک رشته خاتمه یافته با **null** و ذخیره شده در آرایه کاراکتری **phrase** می‌کند (خطوط 33-27). بخاطر داریدکه نام آرایه کاراکتری معادل یک اشاره‌گر به اولین کاراکتر آرایه است، از اینرو ارسال **phrase** بعنوان یک آرگومان به **convertToUppercase** ممکن است. تابع **islower** در خط 29 یک آرگومان کاراکتری دریافت و اگر کاراکتر یک حرف کوچک باشد مقدار **true** و در غیراینصورت **false** برگشت می‌دهد.

```
1  // Fig. 8.10: fig08_10.cpp
2  // Converting lowercase letters to uppercase letters
3  // using a non-constant pointer to non-constant data.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  #include <cctype> // prototypes for islower and toupper
9  using std::islower;
10 using std::toupper;
11
12 void convertToUppercase( char * );
13
14 int main()
15 {
16    char phrase[] = "characters and $32.98";
17
18    cout << "The phrase before conversion is: " << phrase;
19    convertToUppercase( phrase );
20    cout << "\nThe phrase after conversion is:  " << phrase << endl;
21    return 0; // indicates successful termination
22 } // end main
23
24 // convert string to uppercase letters
25 void convertToUppercase( char *sPtr )
26 {
27    while ( *sPtr != '\0' ) // loop while current character is not '\0'
28    {
29       if ( islower( *sPtr ) ) // if character is lowercase,
30          *sPtr = toupper( *sPtr ); // convert to uppercase
31
32       sPtr++; // move sPtr to next character in string
33    } // end while
34 } // end function convertToUppercase
```

```
the phrase before conversion is: characters and $32.98
the phrase after conversion is:  CHARACTERS AND $32.98
```

شکل ۱۰-۸ | تبدیل رشته به حرف بزرگ.

کاراکترهای قرار گرفته در محدودهٔ '**a**' تا '**z**' به حروف متناظر و بزرگ خود توسط تابع **toupper** تبدیل می‌شوند (خط 30) و کاراکترهای دیگر بلاتغییر باقی می‌مانند. تابع **toupper** یک کاراکتر بعنوان آرگومان دریافت می‌کند. اگر کاراکتر یک حرف کوچک باشد، حرف بزرگ متناظر با آن برگشت داده می‌شود، در غیر اینصورت کاراکتر اصلی برگردانده خواهد شد. تابع **toupper** و تابع **islower** بخشی از کتابخانه رسیدگی‌کننده به کاراکتر <**cctype**> هستند. پس از پردازش یک کاراکتر، خط 32 مقدار **sPtr**

را یک واحد افزایش می‌دهد (اگر **sPtr** بصورت **const** اعلان شده بود انجام اینکار ممکن نبود). به هنگام اعمال عملگر + به اشاره‌گری که به یک آرایه اشاره می‌کند، آدرس حافظه ذخیره شده در اشاره‌گر برای اشاره به عنصر بعدی آرایه تغییر می‌یابد (در این مورد کاراکتر بعدی در رشته). افزودن یک واحد به اشاره‌گر یک عملیات معتبر در حساب اشاره‌گر است که در بخش ۸–۸ و ۹–۸ با جزئیات آنها بیشتر آشنا خواهید شد.

اشاره‌گر غیرثابت به داده ثابت

یک اشاره‌گر غیرثابت به داده ثابت اشاره‌گری است که می‌تواند برای اشاره به هر ایتم داده از نوع مقتضی تغییر داده شود، اما داده‌ی که به آن اشاره می‌کند در مدت زمان اشاره به آن قابل تغییر نمی‌باشد. می‌توان از چنین اشاره‌گری در دریافت یک آرگون آرایه به تابعی که هر عنصر آرایه را پردازش خواهد کرد استفاده کرد، اما نبایستی اجازه تغییر در داده صادر شود. برای مثال تابع **printCharacters** (خطوط 22-26 از برنامه شکل ۱۱–۸) پارامتر **sPtr** در خط 22 را از نوع **const char *** اعلان کرده است، از اینروست که می‌تواند رشته خاتمه یافته با **null** و بر پایه اشاره‌گر را دریافت کند. مفهوم این اعلان از سمت چپ به این مضمون است «**sPtr** اشاره‌گری به یک ثابت کاراکتری است.» بدنه تابع از یک عبارت **for** (خط 24-25) برای چاپ هر کاراکتر موجود در رشته تا رسیدن به کاراکتر **null** استفاده کرده است. پس از چاپ هر کاراکتر، اشاره‌گر **sPtr** برای اشاره به کاراکتر بعدی در رشته افزایش داده می‌شود (انجام اینکار ممکن است چرا که اشاره‌گر **const** نیست). تابع **main** آرایه **phrase** از نوع **char** را برای ارسال به **printCharacters** ایجاد می‌کند. مجدداً، می‌توانیم آرایه **phrase** را به **printCharacters** ارسال کنیم چرا که نام آرایه واقعاً یک اشاره‌گر به اولین کاراکتر در آرایه است.

```
1  // Fig. 8.11: fig08_11.cpp
2  // Printing a string one character at a time using
3  // a non-constant pointer to constant data.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  void printCharacters( const char * ); // print using pointer to const data
9
10 int main()
11 {
12    const char phrase[] = "print characters of a string";
13
14    cout << "The string is:\n";
15    printCharacters( phrase ); // print characters in phrase
16    cout << endl;
17    return 0; // indicates successful termination
18 } // end main
19
20 // sPtr can be modified, but it cannot modify the character to which
21 // it points, i.e., sPtr is a "read-only" pointer
22 void printCharacters( const char *sPtr )
23 {
24    for ( ; *sPtr != '\0'; sPtr++ ) // no initialization
25       cout << *sPtr; // display character without modification
26 } // end function printCharacters
```

```
The string is:
print characters of a string
```

شکل ۱۱-۸ | چاپ یک کاراکتر در هر زمان با استفاده از یک اشاره‌گر غیرثابت به داده ثابت.

برنامه شکل ۱۲-۸ به توصیف پیغام خطاهای کامپایلر در زمانیکه مبادرت به کامپایل تابعی می‌کند که یک اشاره‌گر غیرثابت به داده ثابت دریافت می‌کند و سپس سعی می‌نماید تا با استفاده از آن اشاره‌گر مبادرت به تغییر دادن داده کند. [نکته: توجه نمائید که پیغام خطای کامپایلر در میان کامپایلرها باهم تفاوت دارد.]

```
1  // Fig. 8.12: fig08_12.cpp
2  // Attempting to modify data through a
3  // non-constant pointer to constant data.
4
5  void f( const int * ); // prototype
6
7  int main()
8  {
9     int y;
10
11    f( &y ); // f attempts illegal modification
12    return 0; // indicates successful termination
13 } // end main
14
15 // xPtr cannot modify the value of constant variable to which it points
16 void f( const int *xPtr )
17 {
18    *xPtr = 100; // error: cannot modify a const object
19 } // end function f
```

Borland C++ command-line compiler error message:

```
Error E2024 fig08_12.cpp 18:
   Cannot modify a const object in function f(const int *)
```

Microsoft Visual C++ .NET compiler error message:

```
c:\cpphtp5_examples\ch08\Fig08_12\fig08_12.cpp(18) :
   error C2166: 1-value specifies const object
```

GNU C++ compiler error message:

```
Fig08_12.cpp: In function void f(const int*)':
Fig08_12.cpp:18: error: assignment of read-only location
```

شکل ۱۲-۸ | مبادرت به تغییر داده از طریق یک اشاره‌گر غیرثابت به داده ثابت.

همانطوری‌که می‌دانید، آرایه‌ها نوع داده‌های به‌هم پیوسته هستند که ایتم‌های داده مرتبط از نوع یکسان را تحت یک نام ذخیره می‌سازند. زمانیکه تابعی با یک آرایه بعنوان آرگومان فراخوانی می‌شود، آرایه به روش مراجعه به تابع ارسال می‌گردد. با این وجود، شی‌ها همیشه به روش مقدار ارسال می‌گردند. یک کپی از کل شی ارسال می‌شود. انجام اینکار مستلزم صرف زمان برای تهیه کپی از هر ایتم داده در شی و ذخیره‌سازی آن در پشته فراخوانی تابع است. زمانیکه باید یک شی به تابعی ارسال شود، می‌توانیم از یک اشاره‌گر به ثابت داده (یا مراجعه به یک ثابت داده) برای بدست آوردن کارایی ارسال به روش مراجعه استفاده کرده و مانع از ارسال به روش مقدار شویم. زمانیکه یک اشاره‌گر به یک شی ارسال می‌گردد، فقط یک کپی از آدرس آن شی تهیه می‌شود و خود شی کپی نمی‌شود. در یک ماشین با آدرس‌های چهار بایتی، یک کپی از چهار بایت حافظه بسیار سریعتر از کپی کردن یک شی بزرگ صورت می‌گیرد.

اشاره‌گر ثابت به داده غیرثابت

یک اشاره‌گر ثابت به داده غیرثابت، اشاره‌گری است که همیشه به همان مکان حافظه اشاره دارد، داده موجود در آن مکان می‌تواند از طریق اشاره‌گر تغییر یابد. اینحالت برای نام آرایه حالت پیش فرض است. نام آرایه یک اشاره‌گر ثابت برای نشان دادن ابتدای آرایه است. به کل داده‌های آرایه می‌توان با نام آرایه و شاخص دسترسی پیدا کرده و آنها را تغییر داد از یک اشاره‌گر ثابت به یک داده غیرثابت می‌توان برای بازیابی یک آرایه بعنوان یک آرگومان به تابعی که به عناصر آرایه با استفاده از شاخص دسترسی پیدا می‌کند، استفاده کرد. باید اشاره‌گرهای که بصورت **const** اعلان می‌شوند در زمان اعلان مقداردهی اولیه گردند. برنامه شکل ۱۳-۸ مبادرت تغییر در یک اشاره‌گر ثابت می‌کند. خط 11 اشاره‌گر **ptr** را بصورت **int * const** اعلان کرده است. این اعلان از سمت راست به چپ به اینصورت خوانده می‌شود «**ptr** یک اشاره‌گر ثابت به مقدار صحیح غیرثابت است.» اشاره‌گر با آدرس متغیر صحیح **x** مقداردهی اولیه شده است. در خط 14 مبادرت به تخصیص آدرس **y** به **ptr** می‌شود، اما کامپایلر یک پیغام خطا تولید می‌کند. توجه کنید که تا رسیدن به خط 13 و تخصیص مقدار 7 به **ptr** * خطا رخ نمی‌دهد.

```
1  // Fig. 8.13: fig08_13.cpp
2  // Attempting to modify a constant pointer to non-constant data.
3
4  int main()
5  {
6     int x, y;
7
8     // ptr is a constant pointer to an integer that can
9     // be modified through ptr, but ptr always points to the
10    // same memory location.
11    int * const ptr = &x; // const pointer must be initialized
12
13    *ptr = 7; // allowed: *ptr is not const
14    ptr = &y; // error: ptr is const; cannot assign to it a new address
15    return 0; // indicates successful termination
16 } // end main
```

Borland C++ command-line compiler error message:

```
Error E2024 fig08_13.cpp 14:Cannot modify a const object in function
main()s
```

Microsoft Visual C++ .NET compiler error message:

```
c:\cpphtp5e_examples\ch08\Fig08_13\fig08_13.cpp(14) : error C2166:
   l-value specifies const object
```

GNU C++ compiler error message:

```
Fig08_13.cpp: In function int main()':
Fig08_13.cpp:14: error: assignment of read-only variable ptr'
```

شکل ۱۳-۸ | اقدام به تغییر یک اشاره‌گر ثابت به داده غیرثابت.

خطای برنامه‌نویسی

*عدم مقداردهی اولیه اشاره‌گری که بصورت **const** اعلان شده، خطای کامپایلر بدنبال خواهد داشت.*

اشاره‌گر ثابت به داده ثابت

آخرین مجوز دسترسی توسط یک اشاره‌گر ثابت به داده ثابت اعطا می‌شود. همیشه چنین اشاره‌گری به همان موقعیت حافظه اشاره می‌کند و داده موجود در آن مکان از حافظه با استفاده از اشاره‌گر قادر به تغییر نمی‌باشد. در اینحالت است که آرایه به تابعی ارسال می‌شود و آن تابع فقط می‌تواند با استفاده از شاخص،

آرایه را خوانده و قادر به تغییر آرایه نخواهد بود. در برنامه شکل ۱۴-۸ یک متغیر اشاره‌گر بنام ptr از نوع const int * const اعلان شده است (خط 14). اگر این اعلان از سمت راست به چپ خوانده شود به اینصورت خواهد بود «ptr یک اشاره‌گر ثابت به یک ثابت صحیح است.» در خروجی برنامه پیغام‌های خطای تولید شده به هنگام مبادرت به اعمال تغییر در داده‌ای که ptr به آن اشاره می‌کند و (خط 18) در زمان اعمال تغییر در آدرس ذخیره شده در متغیر اشاره‌گر (خط 19) دیده می‌شود.

```
1  // Fig. 8.14: fig08_14.cpp
2  // Attempting to modify a constant pointer to constant data.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     int x = 5, y;
10
11    // ptr is a constant pointer to a constant integer.
12    // ptr always points to the same location; the integer
13    // at that location cannot be modified.
14    const int *const ptr = &x;
15
16    cout << *ptr << endl;
17
18    *ptr = 7; // error: *ptr is const; cannot assign new value
19    ptr = &y; // error: ptr is const; cannot assign new address
20    return 0; // indicates successful termination
21 } // end main
```

Borland C++ command-line compiler error message:

```
Error E2024 fig08_14.cpp 18: Cannot modify a const object in function main()
Error E2024 fig08_14.cpp 19: Cannot modify a const object in function main()
```

Microsoft Visual C++ .NET compiler error message:

```
c:\cpphtp5e_examples\ch08\Fig08_13\fig08_14.cpp(18) : error  C2166:
   l-value specifies const object
c:\cpphtp5e_examples\ch08\Fig08_13\fig08_14.cpp(19) : error  C2166:
   l-value specifies const object
```

GNU C++ compiler error message:

```
Fig08_14.cpp: In function 'int main()':
Fig08_14.cpp:18: error: assignment of read-only location
Fig08_14.cpp:19: error: assignment of read-only variable 'ptr'
```

شکل ۱۴-۸ | اقدام به تغییر یک اشاره‌گر ثابت به داده ثابت.

۶-۸ مرتب سازی انتخابی به روش مراجعه

در این بخش، اقدام به تعریف برنامه‌ای می‌کنیم که به توصیف نحوه ارسال آرایه‌ها و عناصر جداگانه آن به روش مراجعه می‌پردازد. از الگوریتم *مرتب‌سازی انتخابی* استفاده می‌کنیم که برنامه آسانی است، اما الگوریتم آن از کارایی پایانی برخوردار است. در اولین تکرار، الگوریتم کوچکترین عنصر در آرایه را انتخاب کرده و آنرا با اولین عنصر جابجا می‌کند. در دومین تکرار، دومین عنصر کوچکتر (که کوچکترین در میان باقیمانده عناصر است) انتخاب شده و با دومین عنصر تعویض می‌شود. الگوریتم به کار خود ادامه می‌دهد تا اینکه در آخرین تکرار دومین عنصر بزرگ را انتخاب و با آنرا با شاخص دومین عنصر کوچک

عوض کرده، عنصر بزرگ را در آخرین شاخص نگه می‌دارد. پس از i^{th} تکرار، کوچکترین ایتم‌های i از آرایه به ترتیب صعودی مرتب خواهند شد. برای مثال به آرایه زیر توجه کنید

```
34  56  4  10  77  51  93  30  5  52
```

برنامه‌ای که مرتب‌سازی انتخابی را پیاده‌سازی می‌کند ابتدا مبادرت به تعیین کوچکترین عنصر (4) در این آرایه می‌کند که در عنصر دوم جای دارد. برنامه جای 4 را با عنصر صفر (34) عوض می‌کند و در نتیجه لیست زیر خواهد بود

```
4  56  34  10  77  51  93  30  5  52
```

سپس برنامه کوچکترین مقدار باقیمانده در میان سایر عناصر (همه عناصر بجز 4) را که 5 باشد پیدا می‌کند که در عنصر هشتم جای دارد. برنامه جای 5 را با عنصر یک (56) عوض می‌کند و نتیجه کار لیست زیر است

```
4  5  34  10  77  51  93  30  56  52
```

در تکرار سوم، برنامه کوچکترین مقدار بعدی را تعیین (10) و جای آنرا با عنصر دوم (34) عوض می‌کند

```
4  5  10  34  77  51  93  30  56  52
```

این فرآیند تا مرتب شدن کامل آرایه ادامه می‌یابد.

```
4  5  10  30  34  51  52  56  77  93
```

دقت کنید که پس از اولین تکرار، کوچکترین عنصر در اولین مکان جای خواهد گرفت. پس از دومین تکرار، دو عنصر کوچکتر به ترتیب در دو موقعیت اول جای خواهند گرفت. پس از سومین تکرار، سه عنصر کوچکتر به ترتیب در سه موقعیت اول جای خواهند گرفت.

برنامه شکل ۱۵-۸ مبادرت به پیاده‌سازی الگوریتم مرتب‌سازی انتخابی با استفاده از دو تابع **selectionSort** و **swap** کرده است. تابع **selectionSort** در خطوط 53-36 آرایه را مرتب می‌کند. در خط 38 متغیر **smallest** اعلان شده که شاخص کوچکترین عنصر در آرایه باقیمانده را در خود ذخیره می‌کند. خطوط 52-41 حلقه‌ای به تعداد **size - 1** بوجود می‌آورند. خط 43 مبادرت به تنظیم شاخص کوچکترین عنصر با شاخص جاری می‌کند. خطوط 49-46 حلقه‌ای بر روی باقیمانده عناصر در آرایه تدارک می‌بینند. برای هر کدامیک از این عناصر، خط 48 مقدار خود را با مقدار کوچکترین عنصر مقایسه می‌کند. اگر عنصر جاری کوچکتر از کوچکترین عنصر باشد، خط 49 مبادرت به تخصیص شاخص عنصر جاری با **smallest** می‌کند. زمانیکه این حلقه پایان پذیرد، **smallest** حاوی شاخص کوچکترین عنصر در باقیمانده آرایه خواهد بود. خط 51 تابع **swap** را برای قرار دادن کوچکترین عنصر باقیمانده در نقطه بعدی آرایه فراخوانی می‌کند، یعنی مبادله عناصر آرایه **array[smallest]**و**array[i]** .

```
1  // Fig. 8.15: fig08_15.cpp
2  // This program puts values into an array, sorts the values into
3  // ascending order and prints the resulting array.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
```

```
8  #include <iomanip>
9  using std::setw;
10
11 void selectionSort( int * const, const int ); // prototype
12 void swap( int * const, int * const ); // prototype
13
14 int main()
15 {
16    const int arraySize = 10;
17    int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
18
19    cout << "Data items in original order\n";
20
21    for ( int i = 0; i < arraySize; i++ )
22       cout << setw( 4 ) << a[ i ];
23
24    selectionSort( a, arraySize ); // sort the array
25
26    cout << "\nData items in ascending order\n";
27
28    for ( int j = 0; j < arraySize; j++ )
29       cout << setw( 4 ) << a[ j ];
30
31    cout << endl;
32    return 0; // indicates successful termination
33 } // end main
34
35 // function to sort an array
36 void selectionSort( int * const array, const int size )
37 {
38    int smallest; // index of smallest element
39
40    // loop over size - 1 elements
41    for ( int i = 0; i < size - 1; i++ )
42    {
43       smallest = i; // first index of remaining array
44
45       // loop to find index of smallest element
46       for ( int index = i + 1; index < size; index++ )
47
48          if ( array[ index ] < array[ smallest ] )
49             smallest = index;
50
51       swap( &array[ i ], &array[ smallest ] );
52    } // end if
53 } // end function selectionSort
54
55 // swap values at memory locations to which
56 // element1Ptr and element2Ptr point
57 void swap( int * const element1Ptr, int * const element2Ptr )
58 {
59    int hold = *element1Ptr;
60    *element1Ptr = *element2Ptr;
61    *element2Ptr = hold;
62 } // end function swap
```

```
Data item in original order
   2    6    4    8    10    12    89    68    45    37
Data item in ascending order
   2    6    4    8    10    12    37    45    68    89
```

شکل ۱۵-۸ | مرتب سازی انتخابی به روش مراجعه.

اجازه دهید نگاهی دقیق‌تر به تابع **swap** داشته باشیم. بخاطر دارید که C++ تاکید بر پنهان سازی اطلاعات مابین توابع دارد، از اینرو **swap** مجبور به دسترسی به عناصر جداگانه آرایه در **selectionSort** ندارد. چرا که **selectionSort** می‌خواهد **swap** به عناصر آرایه که جابجا یا تعویض خواهند شد دسترسی داشته

باشد. تابع **selectionSort** هر کدامیک از این عناصر را به **swap** به روش مراجعه ارسال می‌کند، آدرس هر عنصر آرایه بصورت صریح ارسال می‌شود. اگر چه کل آرایه‌ها به روش مراجعه ارسال می‌شوند، عناصر مجزای آرایه حالت اسکالر دارند و معمولاً به روش مقدار ارسال می‌شوند. از اینرو، تابع **selectionSort** از عملگر آدرس (&) بر روی هر یک از عناصر آرایه در فراخوانی **swap** استفاده کرده (خط 51) تا تاثیر ارسال با مراجعه بوجود آید. تابع **swap** در خطوط 62-57 مبادرت به دریافت **&array[i]** بشکل متغیر اشاره‌گر **element1Ptr** می‌کند. پنهان‌سازی اطلاعات از دانستن نام **array[i]** جلوگیری می‌کند، اما swap می‌تواند با استفاده از **element1Ptr*** بعنوان مترادفی برای **array[i]** استفاده کند.

از اینرو، زمانیکه **swap** به **element1Ptr*** مراجعه می‌کند در واقع به **array[i]** در **selectionSort** مراجعه می‌نماید. به همین ترتیب، هنگامی‌که **swap** به **element2Ptr*** مراجعه می‌کند در واقع به **array[smallest]** در **selectionSort** مراجعه می‌نماید.

ولو اینکه **swap** اجازه استفاده از عبارات

```
hold = array[i];
array[i] = array[smallest];
array[smallest] = hold;
```

را ندارد، می‌توان همان کارها را با عبارت زیر در تابع **swap** برنامه ۱۵-۸ انجام داد:

```
int hold = *element1Ptr;
*element1Ptr = *element2Ptr;
*element2Ptr = hold;
```

باید به چندین ویژگی تابع **selectionSort** توجه کنید. سرآیند تابع (خط 36) مبادرت به اعلان **array** بصورت **int *const array** بجای **int array[ ]** کرده تا نشان دهد که تابع **selectionSort** یک آرایه تک بعدی بعنوان آرگومان دریافت می‌کند. هر دو پارامتر اشاره‌گر **array** و پارامتر **size** بصورت **const** اعلان شده‌اند تا آخرین قاعده مجوز دسترسی اعمال شود. اگر چه پارامتر **size** یک کپی از مقدار در **main** را دریافت می‌کند و تغییر در کپی نمی‌تواند مقدار را در **main** تغییر دهد، **selectionSort** نیاز به تغییر دادن **size** برای انجام وظیفه خود ندارد. سایز آرایه در مدت زمان اجرای تابع **selectionSort** بلاتغییر باقی می‌ماند. از اینرو، **size** بصورت **const** اعلان شده تا مطمئن گردیم که تغییر پیدا نخواهد کرد. اگر سایز آرایه در زمان اجرای فرآیند مرتب سازی تغییر یابد، الگوریتم مرتب‌سازی بدرستی کار نخواهد کرد. توجه نمائید که تابع **selectionSort** سایز آرایه را بعنوان یک پارامتر دریافت می‌کند چرا که تابع بایستی اطلاعاتی برای مرتب‌سازی آرایه در اختیار داشته باشد. زمانیکه یک آرایه مبتنی بر اشاره‌گر به تابع ارسال می‌شود، فقط آدرس حافظه اولین عنصر آرایه توسط تابع دریافت می‌شود، و باید سایز آرایه هم بصورت مجزا به تابع ارسال گردد.

با تعریف تابع **selectionSort** برای دریافت سایز آرایه بعنوان یک پارامتر، تابعی خواهیم داشت که می‌توان از آن در هر برنامه‌ای که آرایه‌های **int** یک بعدی با هر سایز را مرتب می‌کند، استفاده کنیم. می‌توان سایز آرایه را مستقیماً در تابعی بصورت برنامه‌نویسی شده بدست آورد، اما اینکار می‌تواند تابع را محدود به پردازش آرایه‌ای با سایز مشخص کرده و از کارایی و استفاده مجدداً آن کم کند.

مهندسی نرم‌افزار

به هنگام ارسال آرایه به یک تابع، سایز آرایه را هم ارسال کنید. تا کارایی و استفاده مجدد از تابع افزایش یابد.

۸-۷ عملگر sizeof

زبان C++ دارای عملگر غیربایتری ***sizeof*** برای تعیین سایز یک آرایه (یا نوع داده‌های دیگر، متغیر یا ثابت) برحسب بایت در زمان کامپایل برنامه است. زمانیکه بر روی نام یک آرایه همانند شکل ۱۶-۸ (خط 14) اعمال می‌شود، این عملگر مجموع تعداد بایت‌های موجود در آرایه را بعنوان مقداری از نوع **size_t** (نام مستعار برای **unsigned int** بر روی اکثر کامپایلرها) برگشت می‌دهد. دقت کنید که این نوع متفاوت از **size** در **vector<int>** است که تعداد عناصر صحیح در بردار را مشخص می‌کند. کامپیوتری که ما از آن برای کامپایل کردن این برنامه استفاده کرده‌ایم، متغیرها از نوع **double** را در 8 بایت حافظه ذخیره می‌سازد و آرایه اعلان شده با 20 عنصر (خط 12) از 160 بایت حافظه استفاده می‌کند. زمانیکه یک پارامتر اشاره‌گر (خط 24) را در تابعی استفاده می‌کنیم که یک آرایه را بعنوان آرگومان دریافت می‌کند، عملگر **sizeof** سایز اشاره‌گر را برحسب بایت (4) و نه سایز آرایه برگشت می‌دهد.

```
1  // Fig. 8.16: fig08_16.cpp
2  // Sizeof operator when used on an array name
3  // returns the number of bytes in the array.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  size_t getSize( double * ); // prototype
9
10 int main()
11 {
12    double array[ 20 ]; // 20 doubles; occupies 160 bytes on our system
13
14    cout << "The number of bytes in the array is " << sizeof( array );
15
16    cout << "\nThe number of bytes returned by getSize is "
17       << getSize( array ) << endl;
18    return 0; // indicates successful termination
19 } // end main
20
21 // return size of ptr
22 size_t getSize( double *ptr )
23 {
24    return sizeof( ptr );
25 } // end function getSize
```

```
The number of bytes in the array is 160
The number of bytes returned by getSize is 4
```

شکل ١٦–٨ | اعمال عملگر sizeof بر روی نام آرایه برای برگشت دادن تعداد بایت‌ها در آرایه.

البته می‌توان تعداد عناصر در آرایه را با استفاده از نتایج بدست آمد از دوبار اعمال **sizeof** تعیین کرد. برای مثال، به اعلان آرایه زیر توجه کنید:

```
double realArray[22];
```

اگر متغیرها از نوع داده **double** در هشت بایت حافظه ذخیره شوند، آرایه **realArray** در مجموع 76 بایت خواهد بود. برای تعیین تعداد عناصر در آرایه می‌توان از عبارت زیر استفاده کرد:

```
sizeof realArray / sizeof(double) // calculate number of elements
```

این عبارت تعیین کننده تعداد بایت‌ها در آرایه **realArray** بوده و آن مقدار را بر تعداد بایت‌های بکار رفته در حافظه برای ذخیره یک مقدار **double** تقسیم کرده و نتیجه کار تعداد عناصر در **realArray** یعنی 22 خواهد بود.

تعیین سایز داده‌های بنیادین، آرایه و اشاره‌گر

در برنامه شکل ۱۷–۸ از عملگر **sizeof** برای محاسبه تعداد بایت‌های بکار رفته در ذخیره انواع داده‌های استاندارد استفاده شده است. توجه کنید که، در خروجی برنامه نوع‌های **double** و **long double** دارای سایز یکسان هستند. انواع داده براساس نوع سیستم دارای سایزهای مختلفی می‌باشند.

```
1  // Fig. 8.17: fig08_17.cpp
2  // Demonstrating the sizeof operator.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     char c; // variable of type char
10    short s; // variable of type short
11    int i; // variable of type int
12    long l; // variable of type long
13    float f; // variable of type float
14    double d; // variable of type double
15    long double ld; // variable of type long double
16    int array[ 20 ]; // array of int
17    int *ptr = array; // variable of type int *
18
19    cout << "sizeof c = " << sizeof c
20       << "\tsizeof(char) = " << sizeof( char )
21       << "\nsizeof s = " << sizeof s
22       << "\tsizeof(short) = " << sizeof( short )
23       << "\nsizeof i = " << sizeof i
24       << "\tsizeof(int) = " << sizeof( int )
25       << "\nsizeof l = " << sizeof l
26       << "\tsizeof(long) = " << sizeof( long )
27       << "\nsizeof f = " << sizeof f
28       << "\tsizeof(float) = " << sizeof( float )
29       << "\nsizeof d = " << sizeof d
30       << "\tsizeof(double) = " << sizeof( double )
31       << "\nsizeof ld = " << sizeof ld
32       << "\tsizeof(long double) = " << sizeof( long double )
33       << "\nsizeof array = " << sizeof array
34       << "\nsizeof ptr = " << sizeof ptr << endl;
35    return 0; // indicates successful termination
36 } // end main
```

```
sizeof  c = 1       sizeof(char) = 1
```

```
sizeof  s = 1       sizeof(short) = 2
sizeof  i = 4       sizeof(int) = 4
sizeof  l = 4       sizeof(long) = 4
sizeof  f = 4       sizeof(float) = 4
sizeof  d = 8       sizeof(double) = 8
sizeof  ld = 8      sizeof(long double) = 8
sizeof  array = 80
sizeof  ptr - 4
```

شکل ۱۷-۸ | عملگر sizeof در تعیین سایز انواع داده استاندارد.

می‌توان عملگر sizeof را بر روی نام هر متغیر، نام نوع یا مقدار ثابت اعمال کرد. زمانیکه sizeof بر روی نام یک متغیر یا مقدار ثابت بکار برده می‌شود (که نام یک آرایه نیست) تعداد بایت‌های بکار رفته در ذخیره سازی آن نوع متغیر یا ثابت برگشت داده می‌شود. دقت کنید که استفاده از پرانتز در sizeof در صورتی لازم خواهد بود که نام نوع بعنوان عملوند بکار رفته باشد (همانند int). استفاده از پرانتز در کنار sizeof زمانیکه عملوند sizeof نام یک متغیر یا ثابت باشد، ضروری نیست. بخاطر دارید که sizeof یک عملگر است نه تابع و به همین دلیل تاثیر آن در زمان کامپایل و نه اجرا مشخص می‌شود.

خطای برنامه‌نویسی

*نادیده گرفتن پرانتزها در **sizeof** زمانیکه عملوند نام نوع است، خطای کامپایل بدنبال خواهد داشت.*

کارائی

*بدلیل اینکه **sizeof** یک عملگر غیرباینری زمان کامپایل است نه عملگر زمان اجرا، استفاده صحیح از **sizeof** تاثیر منفی در کارایی اجرا نخواهد گذاشت.*

اجتناب از خطا

*برای دوری کردن از خطای مرتبط با حذف پرانتزها در اطراف عملوند عملگر **sizeof** بسیاری از برنامه نویسان ترجیح می‌دهند در اطراف هر عملوند **sizeof** پرانتز قرار دهند.*

۸-۸ عبارات اشاره‌گر و محاسبات اشاره‌گر

در عبارات محاسباتی، عبارات تخصیص‌ی و مقایسه‌ای، حضور اشاره‌گرها بعنوان عملوند معتبر است. با این همه معمولاً حضور تمام عملگرها در چنین عباراتی با متغیرهای اشاره‌گر معتبر نمی‌باشد. در این بخش به توصیف عملگرهای می‌پردازیم که می‌توانند حضور اشاره‌گرها را بعنوان عملوند قبول کرده و همچنین به بررسی نحوه استفاده از این عملگرها در کنار اشاره‌گرها خواهیم پرداخت.

چندین عملیات محاسباتی وجود دارد که می‌توان بر روی اشاره‌گرها اعمال کرد. می‌توان یک اشاره‌گر را افزایش (++) یا کاهش (--) داد، یک مقدار صحیح به اشاره‌گر افزود (=+ یا +) ، یا مقدار صحیح را از اشاره‌گر کم‌کرد (=- یا -) یا یک اشاره‌گر را از دیگری کم کرد.

فرض کنید آرایه **int v[5]** اعلان شده باشد و اولین عنصر آن در حافظه‌ی با موقعیت 3000 جای گرفته باشد. فرض کنید که اشاره‌گر **vPtr** برای اشاره به **v[0]** مقداردهی اولیه شده باشد (یعنی مقدار **vPtr** برابر

3000 باشد). این وضعیت در دیاگرام شکل ۱۸–۸ بر روی ماشینی نشان داده شده که مقادیر صحیح را در چهار بایت نگهداری می‌کند. توجه کنید که **vPtr** می‌تواند برای اشاره به آرایه **v** با عبارات زیر اشاره کند (چرا که نام آرایه معادل با آدرس اولین عنصر آن است):

```
int * vPtr = v;
int *vPtr = &v [0];
```

شکل ۱۸–۸ | آرایه v و متغیر اشاره‌گر vPtr که به v اشاره می‌کند.

در یک محاسبه عادی نتیجه جمع 2 + 3000 مقدار 3002 است. اما چنین جمعی در مورد محاسبات اشاره‌گر صادق نیست. زمانیکه یک مقدار صحیح به اشاره‌گر افزوده یا از آن کاسته می‌شود، اشاره‌گر به سادگی فقط به آن میزان افزایش یا کاهش نمی‌یابد بلکه آن مقدار صحیح در سایزی که اشاره‌گر به آن مراجعه دارد ضرب می‌شود. برای مثال عبارت

```
vPtr += 2;
```

مقدار 3008 را تولید می‌کند (4 * 2 + 3000) با فرض اینکه یک **int** در چهار بایت ذخیره می‌شود. هم‌اکنون در آرایه v، اشاره‌گر **vPtr** به v[2] اشاره خواهد کرد (شکل ۱۹–۸). اگر یک مقدار صحیح در دو بایت از حافظه ذخیره شود، پس محاسبه فوق حافظه‌ای با موقعیت 3004 را بدست خواهد داد (3000 + 2 * 2). اگر **vPtr** به 3016 افزایش داده شده باشد که به v[4] اشاره کند، عبارت

```
vPtr -= 4;
```

مبادرت به تنظیم **vPtr** برای برگشت به 3000 می‌کند، یعنی ابتدای آرایه. اگر اشاره‌گر در هر بار یک واحد افزایش یا کاهش یابد، می‌توان از عملگرهای افزایش دهنده (++) و کاهش‌دهنده (--) استفاده کرد.

شکل ۱۹–۸ | اشاره‌گر vPtr پس از محاسبات اشاره‌گر.

هر کدام از عبارات

```
++vPtr;
vPtr++;
```

سبب افزایش اشاره‌گر برای اشاره به عنصر بعدی در آرایه می‌شوند. هر یک از عبارات زیر

```
--vPtr;
vPtr--;
```

سبب کاهش اشاره‌گر برای اشاره به عنصر قبلی در آرایه می‌شوند.

متغیرهای اشاره‌گر، اشاره کننده به یک آرایه می‌توانند از یکدیگر کم شوند. برای مثال اگر **vPtr** حاوی موقعیت 3000 و **v2Ptr** حاوی آدرس 3008 باشد، عبارت

```
x = v2Ptr - vPtr;
```

تعداد عناصر آرایه از **vPtr** به **v2Ptr** را که در این مورد 2 است به x تخصیص می‌دهد. محاسبات اشاره‌گر در صورتیکه بر روی اشاره‌گری که به یک آرایه اشاره می‌کند اعمال نشود، بی‌معنی خواهد بود.

نمی‌توانیم تصور کنیم که دو متغیر از یک نوع دقیقاً در پشت سرهم در حافظه ذخیره شده باشند مگر اینکه آن دو عناصری از یک آرایه باشند.

خطای برنامه‌نویسی

استفاده از محاسبات اشاره‌گر بر روی اشاره‌گری که به مقادیر یک آرایه اشاره نمی‌کند، خطای منطقی است.

خطای برنامه‌نویسی

کاهش یا مقایسه دو اشاره‌گر که به عناصر یک آرایه اشاره نمی‌کند خطای منطقی است.

اگر هر دو اشاره‌گر از نوع یکسانی باشند، می‌توان یک اشاره‌گر را به دیگری تخصیص داد. در غیر اینصورت باید از یک عملگر **cast** برای تبدیل مقدار اشاره‌گر قرار گرفته در سمت راست تخصیص به نوع اشاره‌گر در سمت راست عملگر تخصیص استفاده کرد. استثنائی که در قانون وجود دارد در ارتباط با اشاره‌گر به **void** است (یعنی *** void**) که یک اشاره‌گر عام بوده و قادر به عرضه هر نوع اشاره‌گر می‌باشد. تمام انواع اشاره‌گر را می‌توان به اشاره‌گری از نوع *** void** تخصیص داد بدون اینکه نیازی به تبدیل باشد. با این همه، یک اشاره‌گر از نوع *** void** نمی‌تواند مستقیماً به یک اشاره‌گر از نوع دیگر تخصیص داده شود. بایستی ابتدا اشاره‌گر از نوع *** void** تبدیل به نوع اشاره‌گر مناسب و صحیح شود.

یک اشاره‌گر از نوع *** void** قادر به ردگیری نیست. برای مثال، کامپایلر می‌داند که یک اشاره‌گر به **int** به چهار بایت از حافظه بر روی ماشینی با مقادیر صحیح چهاربایتی مراجعه دارد، اما یک اشاره‌گر به **void** فقط حاوی یک آدرس حافظه برای یک نوع داده ناشناخته است. کامپایلر برای تعیین تعداد بایت نیاز به دانستن نوع داده دارد تا بتواند یک اشاره‌گر خاص را ردگیری کند.

خطای برنامه‌نویسی

*تخصیص یک اشاره‌گر از یک نوع به اشاره‌گری از نوع دیگر (بجز *** void**) بدون اینکه عمل تبدیل اشاره‌گر اول به نوع اشاره‌گر دوم صورت گرفته باشد، خطای زمان کامپایل بدنبال خواهد داشت.*

با استفاده از عملگرهای تساوی و رابطه‌ای می‌توان به مقایسه اشاره‌گرها پرداخت. مقایسه با استفاده از عملگرهای رابطه‌ای فرآیند بی‌معنی خواهد بود، مگر اینکه اشاره‌گرها در حال اشاره به عضوهای یک آرایه باشند. در مقایسه اشاره‌گرها فرآیند مقایسه آدرسهای ذخیره شده در اشاره‌گرها صورت می‌گیرد. برای مثال در مقایسه دو اشاره‌گر می‌توان تعیین کرد که آیا اشاره‌گری به عنصر بالاتری در آرایه به نسبت اشاره‌گر دیگری اشاره می‌کند یا خیر. یکی از استفاده‌های رایج از عملیات مقایسه اشاره‌گر، تعیین اشاره اشاره‌گر به صفر است (یعنی، اشاره‌گر یک اشاره‌گر ***null*** است و به چیزی اشاره نمی‌کند).

۹-۸ رابطه مابین اشاره‌گرها و آرایه‌ها

آرایه‌ها و اشاره‌گرها رابطه نزدیکی در C++ دارند و تقریباً می‌توان از آنها بجای یکدیگر استفاده کرد. می‌توان در مورد نام آرایه همانند یک اشاره‌گر ثابت فکر کرد. می‌توان از اشاره‌گرها برای انجام هر کاری که توسط شاخص آرایه می‌توان انجام داد، بکار گرفت. اعلان‌های زیر را در نظر بگیرید:

```
int b[5];  // create 5-element int array b
int *bPtr; // create int pointer bPtr
```

بدلیل اینکه نام آرایه (بدون شاخص) یک اشاره‌گر (ثابت) به اولین عنصر آرایه است، می‌توانیم **bPtr** را با آدرس اولین عنصر در آرایه **b** و با عبارت زیر تنظیم کنیم

```
bPtr = b; // assign address of array b to bPtr
```

این عبارت معادل دریافت آدرس اولین عنصر آرایه بصورت زیر است:

```
bPtr = &b[0]; // also assigns address of array b to bPtr
```

به عنصر **b[3]** آرایه می‌توان بطور جایگزین با عبارت اشاره‌گر زیر مراجعه کرد:

```
*(bPtr + 3)
```

عدد 3 در عبارت فوق افست به اشاره‌گر است. زمانیکه اشاره‌گر به ابتدای آرایه اشاره می‌کند، افست بر این نکته دلالت دارد که کدام عنصر آرایه باید مورد مراجعه واقع شود و مقدار افست معادل با شاخص آرایه است. از جمله فوق بعنوان *نشان‌گذاری اشاره‌گر/افست* یاد می‌شود. وجود پرانتزها ضروری است، چرا که اولویت * از + بالاتر است. بدون حضور پرانتز عبارت فوق مبادرت به افزودن 3 به مقدار **bPtr*** خواهدکرد (یعنی 3 به **b[0]** افزوده می‌شود، با فرض اینکه **bPtr** به ابتدای آرایه اشاره می‌کند). همانطوری که عنصر آرایه می‌تواند با یک عبارت اشاره‌گر مورد مراجعه قرار گیرد، آدرس

```
&b[3]
```

می‌تواند با عبارت اشاره‌گر و بصورت زیر نوشته شود

```
bPtr + 3
```

می‌توان با نام آرایه همانند یک اشاره‌گر رفتار کرد و در محاسبات اشاره‌گر بکار گرفت. برای مثال، عبارت

```
*(b + 3)
```

به عنصر **b[3]** آرایه مراجعه دارد. بطور کلی، می‌توان تمام عبارات آرایه که با شاخص انجام می‌شود را با نوشتن اشاره‌گر و یک افست انجام داد. در این مورد، نشانه‌گذاری اشاره‌گر/افست با نام آرایه بعنوان اشاره‌گر بکار گرفته شده است. دقت کنید که عبارت قبلی مبادرت به تغییر نام آرایه نمی‌کند، **b** هنوز هم به اولین عنصر در آرایه اشاره می‌کند.

همانند آرایه‌ها می‌توان اشاره‌گرها را هم شاخص‌دار کرد. برای مثال، عبارت

```
bPtr[1]
```

به عنصر **b[1]** آرایه مراجعه می‌کند، این عبارت از نشان‌گذاری اشاره‌گر/شاخص استفاده کرده است. بخاطر دارید که نام یک آرایه یک اشاره‌گر ثابت است که همیشه به ابتدای آرایه اشاره می‌کند. از اینرو عبارت

```
b += 3
```

سبب بوجود آمدن خطای کامپایل خواهد شد، چرا که این عبارت مبادرت به تغییر مقدار نام آرایه (یک ثابت) با محاسبات اشاره‌گر می‌کند.

خطای برنامه‌نویسی

با اینکه اسامی آرایه‌ها، اشاره‌گرهای به ابتدای آرایه هستند و اشاره‌گرها می‌توانند در عبارات محاسباتی تغییر داده شوند، اما اسامی آرایه‌ها را نمی‌توان در عبارات محاسباتی دچار تغییر کرد، چرا که اسامی آرایه‌ها از جمله اشاره‌گرهای ثابت هستند.

در برنامه شکل ۲۰-۸ از چهار نشان‌گذاری بحث شده در این بخش برای مراجعه به عناصر آرایه برای انجام یک وظیفه که چاپ چهار عناصر آرایه b از نوع صحیح باشد، استفاده شده است.

```
1  // Fig. 8.20: fig08_20.cpp
2  // Using subscripting and pointer notations with arrays.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     int b[] = { 10, 20, 30, 40 }; // create 4-element array b
10    int *bPtr = b; // set bPtr to point to array b
11
12    // output array b using array subscript notation
13    cout << "Array b printed with:\n\nArray subscript notation\n";
14
15    for ( int i = 0; i < 4; i++ )
16       cout << "b[" << i << "] = " << b[ i ] << '\n';
17
18    // output array b using the array name and pointer/offset notation
19    cout << "\nPointer/offset notation where "
20       << "the pointer is the array name\n";
21
22    for ( int offset1 = 0; offset1 < 4; offset1++ )
23       cout << "*(b + " << offset1 << ") = "<< *( b + offset1 )<< '\n';
24
25    // output array b using bPtr and array subscript notation
26    cout << "\nPointer subscript notation\n";
27
28    for ( int j = 0; j < 4; j++ )
29       cout << "bPtr[" << j << "] = " << bPtr[ j ] << '\n';
30
31    cout << "\nPointer/offset notation\n";
32
33    // output array b using bPtr and pointer/offset notation
34    for ( int offset2 = 0; offset2 < 4; offset2++ )
35       cout << "*(bPtr + " << offset2 << ") = "
36          << *( bPtr + offset2 ) << '\n';
37
38    return 0; // indicates successful termination
39 } // end main
```

```
Array b printed with:

Array subscript notation
b[0] =  10
b[1] =  20
b[2] =  30
b[3] =  40
Pointer/offset notation where the pointer is the array name
*(b + 0) = 10
```

```
*(b + 1) = 20
*(b + 2) = 30
*(b + 3) = 40
Pointer subscript notation
bPtr [0] =  10
bPtr [1] =  20
bPtr [2] =  30
bPtr [3] =  40
Pointer/offset notation
*(bPtr + 0) = 10
*(bPtr + 1) = 20
*(bPtr + 2) = 30
*(bPtr + 3) = 40
```

شکل ۲۰-۸ | مراجعه به عناصر آرایه با نام آرایه و اشاره‌گرها.

برای اینکه بهتر متوجه قابلیت تعویض آرایه‌ها و اشاره‌گرها شوید، اجازه دهید تا نگاهی به دو تابع کپی کننده رشته، **copy1** و **copy2** در برنامه ۲۱-۸ داشته باشیم. هر دو تابع مبادرت به کپی یک رشته به یک آرایه کاراکتری می‌کنند. پس از مقایسه نمونه اولیه تابع **copy1** و **copy2** توابع یکسان هستند. این توابع وظیفه یکسانی را انجام می‌دهند اما از پیاده‌سازی متفاوتی برخوردار می‌باشند.

```
1  // Fig. 8.21: fig08_21.cpp
2  // Copying a string using array notation and pointer notation.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  void copy1( char *, const char * ); // prototype
8  void copy2( char *, const char * ); // prototype
9
10 int main()
11 {
12    char string1[ 10 ];
13    char *string2 = "Hello";
14    char string3[ 10 ];
15    char string4[] = "Good Bye";
16
17    copy1( string1, string2 ); // copy string2 into string1
18    cout << "string1 = " << string1 << endl;
19
20    copy2( string3, string4 ); // copy string4 into string3
21    cout << "string3 = " << string3 << endl;
22    return 0; // indicates successful termination
23 } // end main
24
25 // copy s2 to s1 using array notation
26 void copy1( char * s1, const char * s2 )
27 {
28    // copying occurs in the for header
29    for ( int i = 0; ( s1[ i ] = s2[ i ] ) != '\0'; i++ )
30       ; // do nothing in body
31 } // end function copy1
32
33 // copy s2 to s1 using pointer notation
34 void copy2( char *s1, const char *s2 )
35 {
36    // copying occurs in the for header
37    for ( ; ( *s1 = *s2 ) != '\0'; s1++, s2++ )
38       ; // do nothing in body
39 } // end function copy2
```

```
string1 = Hello
string3 = Good Bye
```

شکل ۲۱-۸ | کپی رشته با استفاده از نشان‌گذاری آرایه و اشاره‌گر.

تابع **copy1** (خطوط 31-26) از نشان‌گذاری شاخص آرایه برای کپی رشته در **s2** به آرایه کاراکتری **s1** استفاده کرده است. در این تابع از متغیر شمارنده **i** بعنوان شاخص آرایه استفاده شده است. سرآیند عبارت **for** (خط 29) کل عملیات کپی را انجام می‌دهد، بدنه این عبارت تهی است. سرآیند مشخص می‌کند که **i** با صفر مقداردهی اولیه شده و در هر بار تکرار حلقه یک واحد افزایش می‌یابد. شرط موجود در **for** یعنی **(s1[i] = s2[i]) != '\0'** عملیات کپی کاراکتر به کاراکتر را از **s2** به **s1** انجام می‌دهد. زمانیکه به کاراکتر *null* در **s2** برسد آنرا به **s1** تخصیص داده و حلقه خاتمه می‌پذیرد، چرا که کاراکتر **null** معادل با **'\0'** است. بخاطر دارید که مقدار یک عبارت تخصیصی مقدار تخصیص یافته به عملوند سمت چپ آن است.

تابع **copy2** در خطوط 39-34 از اشاره‌گرها و عبارت محاسباتی اشاره‌گر برای کپی رشته در **s2** به آرایه کاراکتری **s1** استفاده کرده است. مجدداً، سرآیند عبارت **for** در خط 37 کل عملیات کپی را انجام می‌دهد. سرآیند شامل مقداردهی اولیه متغیر نیست. همانند تابع **copy1** شرط، **('\0' != *s2)** عملیات کپی را انجام می‌دهد. اشاره‌گر **s2** ردگیری شده و کاراکتر بدست آمده به اشاره‌گر ردگیری شده **s1** تخصیص داده می‌شود. پس شرط در تخصیص، حلقه هر دو اشاره‌گر را افزایش می‌دهد و از اینرو هر دو بترتیب به عنصر بعدی در آرایه **s1** و کاراکتری بعدی در رشته **s2** اشاره می‌کنند. زمانیکه حلقه با کاراکتر null در **s2** مواجه می‌شود، کاراکتر null به اشاره‌گر ردگیری شده s1 تخصیص داده شده و حلقه خاتمه می‌پذیرد.

آرگومان اول در هر دو تابع **copy1** و **copy2** بایستی برای نگهداری رشته موجود در آرگومان دوم به اندازه کافی بزرگ در نظر گرفته شده باشد. در غیر اینصورت احتمالاً به هنگام اقدام به نوشتن در حافظه‌ای خارج از مرزهای آرایه با خطا مواجه خواهید شد.

۱۰-۸ آرایه‌ای از اشاره‌گرها

آرایه‌ها می‌توانند حاوی اشاره‌گرها هم باشند. استفاده رایج از چنین ساختمان داده‌های بفرم آرایه‌ای از رشته‌های مبتنی بر اشاره‌گر است که گاهاً از آنها بعنوان *آرایه رشته‌ای* یاد می‌شود. هر موجودیت در آرایه یک رشته است، اما در C++ یک رشته ضرورتاً یک اشاره‌گر به اولین کاراکتر خود می‌باشد، از اینرو هر موجودیت در یک آرایه از رشته‌ها یک اشاره‌گر به اولین کاراکتر رشته می‌باشد. به اعلان آرایه رشته‌ای **suit** توجه اینکه که می‌تواند در نمایش مجموعه‌ای از کارت‌های بازی کاربرد داشته باشد:

```
const char *suit[4] =
   {"Hearts" , "Diamonds" , "Clubs" , "Spades"};
```

بخش **suit[4]** از اعلان نشان می‌دهد که آرایه چهار عنصر دارد. بخش **const char *** از اعلان بر این نکته تاکید دارد که هر عنصر از آرایه suit از نوع «اشاره‌گر به داده ثابت char است.» چهار مقدار جای گرفته در آرایه عبارتند از "Hearts"، "Diamonds"، "Clubs" و "Spades". هر کدامیک از آنها در آرایه

بعنوان یک رشته کاراکتری خاتمه یافته با null ذخیره شده‌اند که یک کاراکتر بزرگتر از تعداد کاراکترهای مابین گوتیشن‌ها است. چهار رشته به‌ترتیب دارای طولهای هفت، نه، شش و هفت کاراکتر (با احتساب کاراکتر خاتمه دهنده null) هستند. اگرچه بنظر می‌رسد که این رشته‌ها در آرایه **suit** جای گرفته‌اند، اما در واقع اشاره‌گرها همانند شکل ۲۲-۸ در آرایه ذخیره شده‌اند. هر اشاره‌گر به اولین کاراکتر از رشته متناظر خود اشاره می‌کند. از اینرو، حتی اگر آرایه **suit** دارای سایز ثابت باشد، دسترسی به رشته‌های کاراکتر به هر طول را ممکن می‌سازد. قابلیت انعطاف پذیری در ساختمان‌های داده یکی از توانایی‌های زبان C++ می‌باشد.

شکل ۲۲-۸ | نمایش گرافیکی از آرایه suit .

رشته‌های suit می‌توانند توسط یک آرایه دو بعدی عرضه شوند که در آن هر سطر نشاندهنده یک suit و هر ستون نشاندهنده یکی از حروف نام suit باشد. چنین ساختمان داده‌ای بایستی تعداد ثابت ستون برای هر سطر داشته باشد و آن عدد بایستی بقدر کافی از بزرگترین رشته بزرگ‌تر باشد. بنابر این به هنگام ذخیره تعداد زیادی رشته که اکثر آنها کوتاهتر از بزرگترین رشته هستند، حافظه قابل ملاحظه‌ای به هدر می‌رود. در بخش بعدی با استفاده از آرایه‌ای از رشته‌ها مبادرت به پیاده سازی بازی کارت خواهیم کرد.

معمولاً آرایه‌ای از رشته‌ها به همراه آرگومان‌های خط فرمان بکار گرفته می‌شوند تا در زمان شروع برنامه به تابع **main** ارسال گردند. چنین آرگومان‌های بدنبال اجرای یک برنامه از خط فرمان آورده می‌شوند. غالباً از آرگومان‌های خط فرمان برای ارسال گزینه‌ها به یک برنامه استفاده می‌شود. برای مثال می‌توانیم از آرگومان خط فرمان زیر در ویندوز یاد کنیم:

```
dir /p
```

که سبب لیست شدن محتویات موجود در شاخه جاری شده و پس از پر شدن صفحه، مکث می‌کند. زمانیکه دستور **dir** اجرا می‌شود گزینه **p** به **dir** به عنوان یک آرگومان خط دستور ارسال می‌شود. چنین آرگومان‌های در یک آرایه رشته‌ای که **main** بعنوان آرگومان دریافت می‌کند جای داده می‌شوند.

۱۱-۸ مبحث آموزشی: بازی کارت

در این بخش از روش تولید اعداد تصادفی برای ایجاد یک بازی برزدن کارت و برنامه شبیه‌سازی تقسیم کارت استفاده خواهیم کرد. سپس می‌توان از چنین برنامه‌ای در ایجاد برنامه‌های بازی خاص کارت استفاده کرد. به منظور آشکار شدن برخی از مشکلات کارایی، بطور دانسته از الگوریتم‌های بهینه شده بر زدن و تقسیم کارت استفاده کرده‌ایم.

با استفاده از روش *بالا به پایین، اصلاح گام به گام*، برنامه‌ای ایجاد خواهیم کرد که یک دسته کارت 52 تایی را بر زده و سپس آنها را تقسیم می‌کند. از یک آرایه 4 در 13 دو بعدی بنام **deck** برای نمایش یک دسته کارت بازی استفاده می‌کنیم (شکل ۲۳-۸). سطرها متناظر با مجموعه و به ترتیب سطر صفر با

Hearts، سطر یک با Diamond، سطر دو با Club و سطر سوم با Spade هستند. ستون‌ها متناظر با مقادیر موجود بر روی کارت‌ها هستند، ستون‌ها از 0 تا 9 نشاندهنده مقادیر از 1 (Ace یا تک خال) تا 10 بوده و ستون‌ها از 10 الی 12 به‌ترتیب نشاندهنده Jack (سرباز)، Queen (ملکه) و King (شاه) می‌باشند.

شکل ۲۳-۸ | آرایه دوبعدی برای عرضه یک دسته کارت.

بایستی آرایه رشته‌ای **suit** را با رشته‌های کاراکتری که نشاندهنده چهار مجموعه (همانند شکل ۲۲-۸) می‌باشند پر کرده و آرایه رشته‌ای **face** را با رشته‌های کاراکتری که نشاندهنده 13 مقدار کارت‌ها می‌باشند پر نمائیم.

شبیه‌سازی برزدن یک مجموعه کارت می‌تواند بصورت زیر دنبال شود. ابتدا آرایه **deck** با صفر مقداردهی اولیه می‌شود. سپس، یک سطر از 3-0 و یک ستون از 12-0 بطور تصادفی انتخاب می‌شوند. عدد 1 در عنصر ستون [ستون] [سطر] **deck** وارد می‌شود تا نشان دهد که این کارت در اولین توزیع از برخوردن کارت پخش شده است. این فرآیند با اعداد 2,3,...,52 ادامه یافته و بصورت تصادفی در آرایه **deck** جای می‌گیرند تا نشان دهند که کدام کارت در بر دوم، سوم، ... و پنجاه‌ودوم پخش شده‌اند. همانطوری که آرایه **deck** شروع به پر شدن با اعداد کارت‌ها می‌شوددد، امکان دارد که کارتی دو بار انتخاب شود (یعنی [ستون] [سطر] **deck** به هنگام انتخاب شدن صفر نباشد). چنین انتخابی نادیده گرفته می‌شود و سطرها و ستون‌ها به تکرار و بصورت تصادفی انتخاب می‌شوند تا اینکه یک کارت انتخاب نشده پیدا شود. سرانجام، اعداد 1 الی 52 مبادرت به اشغال 52 شکاف آرایه **deck** می‌کنند. در این نقطه، دسته کارت‌ها کاملاً برخورده‌اند.

اگر کارت‌های که هم اکنون بر زده شده‌اند بطور تصادفی و مکرراً انتخاب شوند، این الگوریتم بر زدن می‌تواند برای یک مدت نامحدود بکار گرفته شود. این پدیده بعنوان تعویق نامعین هم شناخته می‌شود.

پس از توزیع اولین کارت، آرایه را برای یافتن عنصر [ستون] [سطر] **deck** که با 1 مطابقت نماید جستجو می‌کنیم. اینکار با یک دستور **for** تودرتو که سطر آن از 0 تا 3 و ستون آن از 0 تا 12 تغییر می‌کند، قابل انجام است. آرایه **suit** قبلاً با چهار مجموعه پر شده است، از اینرو پس از پیدا کردن کارت، آن مجموعه را گرفته و رشته کاراکتری در [سطر] **suit** را چاپ می‌کنیم. به همین ترتیب، مقدار روی کارت را بدست آورده و رشته کاراکتری در [ستون] **face** را چاپ می‌نمائیم. همچنین رشته "of" را چاپ می‌کنیم. چاپ صحیح و به ترتیب این اطلاعات می‌تواند ما را در چاپ هر کارت بصورت "king of dubs" و "Ace of Diomads" و غیره کمک کند.

اجازه به روش از بالا به پایین، اصلاح گام به گام این برنامه را پردازش کنیم. در بالاترین سطح عبارت زیر قرار دارد

shuffle and deal 52 cards

اولین اصلاح صورت گرفته حاصل زیر را بدست خواهد داد:

```
Initialize the suit array
Initialize the face array
Initialize the deck array
Shuffle the deck
Deal 52 cards
```

می‌توانیم عبارت "Shuffle the deck" را بصورت زیر بسط دهیم:

```
For each of the 52 cards
Place card number in randemly selected unoccupied slot of deck
```

می‌توانیم عبارت "Deal 52 Cards" را بصورت زیر بسط دهیم:

```
For each of 52 cards
  Find card number in deck array and print face and suit of card
```

با در کنار هم قرار دادن این تحلیل‌ها مرحله دوم کامل می‌شود:

```
Initialize the suit array
Initialize the face array
Initialize the dack array
For each of 52 cards
  Place and number in randomly selected unoccupied slot of deck
For each of the 52 cards
  Find card number in deck array and print face and suit of card
```

عبارت "Place card number in randomly selected unoccupied slot of deck" می‌تواند به عبارات زیر بسط داده شود:

```
choose slot of deck randomly
while chosen slot of deck has been previously chosen
  choose slot of deck randomly
place card number in chosen slot of deck
```

عبارت "Find card number in deck array and print face and suit of card" می‌تواند به عبارات زیر بسط داده شود:

```
For each of the 52 cards
    If slot contaius card number
      Print the face and suit of the card
```

با همکاری این عبارات بسط یافته، سومین مرحله اصلاح همانند شکل ۲۴–۸ بدست می‌آید، که فرآیند اصلاح را کامل می‌کند. برنامه‌های شکل ۲۵–۸ الی ۲۷–۸ حاوی برنامه بر زدن و توزیع کارت و یک اجرای نمونه هستند. خطوط 61-67 از تابع **deal** (شکل ۲۶–۸) پیاده سازی کننده خطوط 1-2 از برنامه ۲۴–۸ می‌باشند. سازنده در خطوط 22-35 از شکل ۲۶–۸ پیاده‌کننده خطوط 1-3 از شکل ۲۴–۸ می‌باشد. تابع **shuffle** (خطوط 38-55 از شکل ۲۶–۸) پیاده‌کننده خطوط 5-11 از شکل ۲۴–۸ می‌باشد. تابع **deal** (خطوط 58-88 از شکل ۲۶–۸) پیاده کننده خطوط 13-16 از شکل ۲۴–۸ می‌باشد. به فرمت خروجی بکار رفته در تابع **deal** دقت کنید (خطوط 81-83 از شکل ۲۶–۸).

```
Initialize the suit array
Initialize the face array
Initialize the deck array

For each of 52 cards
```

Choose slot of deck randomly

While slot of deck has been previously chosen
Choose slot of deck randomly

Place card number in chosen slot of deck

For each of the 52 cards
For each of the 52 cards
If slot contaius card number
Print the face and suit of the card

شکل ۲۴-۸ | الگوریتم شبکه متعلق به برنامه توزیع و بر زدن کارت.

```
1  // Fig. 8.25: DeckOfCards.h
2  // Definition of class DeckOfCards that
3  // represents a deck of playing cards.
4
5  // DeckOfCards class definition
6  class DeckOfCards
7  {
8  public:
9     DeckOfCards(); // constructor initializes deck
10    void shuffle(); // shuffles cards in deck
11    void deal(); // deals cards in deck
12 private:
13    int deck[ 4 ][ 13 ]; // represents deck of cards
14 }; // end class DeckOfCards
```

شکل ۲۵-۸ | فایل سرآیند DeckOfCards.

```
1  // Fig. 8.26: DeckOfCards.cpp
2  // Member-function definitions for class DeckOfCards that simulates
3  // the shuffling and dealing of a deck of playing cards.
4  #include <iostream>
5  using std::cout;
6  using std::left;
7  using std::right;
8
9  #include <iomanip>
10 using std::setw;
11
12 #include <cstdlib> // prototypes for rand and srand
13 using std::rand;
14 using std::srand;
15
16 #include <ctime> // prototype for time
17 using std::time;
18
19 #include "DeckOfCards.h" // DeckOfCards class definition
20
21 // DeckOfCards default constructor initializes deck
22 DeckOfCards::DeckOfCards()
23 {
24    // loop through rows of deck
25    for ( int row = 0; row <= 3; row++ )
26    {
27       // loop through columns of deck for current row
28       for ( int column = 0; column <= 12; column++ )
29       {
30          deck[ row ][ column ] = 0; // initialize slot of deck to 0
31       } // end inner for
32    } // end outer for
33
34    srand( time( 0 ) ); // seed random number generator
35 } // end DeckOfCards default constructor
36
37 // shuffle cards in deck
38 void DeckOfCards::shuffle()
39 {
```

```
40    int row; // represents suit value of card
41    int column; // represents face value of card
42
43    // for each of the 52 cards, choose a slot of the deck randomly
44    for ( int card = 1; card <= 52; card++ )
45    {
46       do // choose a new random location until unoccupied slot is found
47       {
48          row = rand() % 4; // randomly select the row
49          column = rand() % 13; // randomly select the column
50       } while( deck[ row ][ column ] != 0 ); // end do...while
51
52       // place card number in chosen slot of deck
53       deck[ row ][ column ] = card;
54    } // end for
55 } // end function shuffle
56
57 // deal cards in deck
58 void DeckOfCards::deal()
59 {
60    // initialize suit array
61    static const char *suit[ 4 ] =
62       { "Hearts", "Diamonds", "Clubs", "Spades" };
63
64    // initialize face array
65    static const char *face[ 13 ] =
66       { "Ace", "Deuce", "Three", "Four", "Five", "Six", "Seven",
67       "Eight", "Nine", "Ten", "Jack", "Queen", "King" };
68
69    // for each of the 52 cards
70    for ( int card = 1; card <= 52; card++ )
71    {
72       // loop through rows of deck
73       for ( int row = 0; row <= 3; row++ )
74       {
75          // loop through columns of deck for current row
76          for ( int column = 0; column <= 12; column++ )
77          {
78             // if slot contains current card, display card
79             if ( deck[ row ][ column ] == card )
80             {
81                cout << setw( 5 ) << right << face[ column ]
82                   << " of " << setw( 8 ) << left << suit[ row ]
83                   << ( card % 2 == 0 ? '\n' : '\t' );
84             } // end if
85          } // end innermost for
86       } // end inner for
87    } // end outer for
88 } // end function deal
```

شکل ۲۶-۸ | تعریف توابع عضو برای برزدن و توزیع کارت.

عبارت خروجی، کارت‌های چهره را با ترازبندی از راست در میدانی به طول پنج کاراکتر و کارت‌های suit را با ترازبندی از چپ در میدانی به طول هشت کاراکتر چاپ می‌کند (شکل ۲۷-۸). خروجی در فرمت دو ستونی چاپ می‌شود.

الگوریتم توزیع کارت دارای یک ضعف است. زمانیکه مطابقتی پیدا می‌شود، حتی اگر این مطابقت در اولین سعی پیدا شود، عبارت **for** تودرتو به جستجو در میان عناصر باقیمانده از کارت‌ها ادامه می‌دهد تا مطابقتی پیدا کند.

```
1  // Fig. 8.27: fig08_27.cpp
2  // Card shuffling and dealing program.
3  #include "DeckOfCards.h" // DeckOfCards class definition
4
```

```
5  int main()
6  {
7     DeckOfCards deckOfCards; // create DeckOfCards object
8
9     deckOfCards.shuffle(); // shuffle the cards in the deck
10    deckOfCards.deal(); // deal the cards in the deck
11    return 0; // indicates successful termination
12 } // end main
```

```
Nine of Spades          Seven of Clubs
Five of Spades          Eight of Clubs
Queen of Diamonds       three of Hearts
Jack of Spades          Five of Diamonds
Jack of Diamonds        Three of Diamonds
Three of Clubs          Six of Clubs
Ten of Clubs            Nine of Diamonds
Ace of Hearts           Queen of Heart
Seven of Spades         Deuce of Spades
Six of Hearts           Deuce of Clubs
Ace of Clubs            Deuce of Diamonds
Nine of Hearts          Seven of Diamonds
Six of Spades           Eight of Diamonds
Ten of Spades           King of Hearts
Four of Clubs           Ace of Spades
Ten of Hearts           Four of Spades
Eight of Hearts         Eight of Spades
Jack of Hearts          Ten of Diamonds
Four of Diamonds        king of Diamonds
Seven of Hearts         King of Spades
Queen of Spades         Four of Hearts
Nine of Clubs           Six of Diamonds
Deuce of Hearts         Jack of Diamonds
King of Clubs           Three of Spades
Queen of Clubs          Five of Clubs
Five of Hearts          Ace of Diamonds
```

شکل ۲۷-۸ | برنامه برزدن و توزیع کارت.

۱۲-۸ اشاره‌گرهای تابع

یک اشاره‌گر به تابع حاوی آدرس تابع در حافظه است. در فصل هفتم، مشاهده کردید که نام یک آرایه در واقع آدرسی در حافظه است که نشاندهنده اولین عنصر آرایه می‌باشد. به همین ترتیب، نام یک تابع در واقع آدرس شروع کدی در حافظه است که وظیفه تابع را به انجام می‌رساند. اشاره‌گرها به توابع می‌توانند به توابع ارسال، از توابع برگشت داده شوند، در آرایه‌ها ذخیره شده و به دیگر اشاره‌گرهای توابع تخصیص داده شوند.

مرتب سازی انتخابی با استفاده از اشاره‌گرهای تابع

برای توضیح نحوه استفاده از اشاره‌گرهای تابع، برنامه شکل ۲۸-۸ را که تغییر یافته برنامه مرتب‌سازی انتخابی است را بکار گرفته‌ایم. برنامه ۲۸-۸ حاوی **main** (خطوط 55-17) و توابع **selectionSort** در خطوط 76-59، **swap** در خطوط 85-80، **ascending** در خطوط 92-89 و **descending** در خطوط 96-99 است. تابع **selectionSort** یک اشاره‌گر به تابع دریافت می‌کند (خواه تابع **ascending** باشد یا تابع **descending**). توابع **ascending** و **descending** تعیین‌کننده ترتیب مرتب‌سازی هستند (صعودی یا نزولی). برنامه به کاربر اعلان می‌کند تا ترتیب مرتب‌سازی آرایه را مشخص کند (خطوط 26-24). اگر کاربر عدد 1 را وارد سازد، اشاره‌گر به تابع **ascending** به تابع **selectionSort** (خط 37) ارسال می‌شود و سبب می‌گردد تا آرایه بصورت صعودی مرتب شود. اگر کاربر عدد 2 را وارد سازد، اشاره‌گر به تابع

descending به تابع **selectionSort** (خط 45) ارسال می‌شود و سبب می‌شود تا آرایه بصورت نزولی مرتب گردد.

پارامتر زیر در خط 60 از سرآیند تابع **selectionSort** ظاهر شده است:

```
bool ( *compare )( int, int )
```

این پارامتر تصریح‌کننده یک اشاره‌گر به تابع است. کلمه کلیدی **bool** بر این نکته دلالت دارد که تابع اشاره کننده یک مقدار بولی برگشت خواهد داد. کلمه **(*compare)** نشاندهنده نام اشاره‌گر به تابع است (* نشان می‌دهد که پارامتر **compare** یک اشاره‌گر است). عبارت **(int , int)** نشان می‌دهد که تابع اشاره شده برای مقایسه، دو آرگومان صحیح دریافت می‌کند. وجود پرانتزها در اطراف ***compare** لازم بوده و نشان می‌دهند که **compare** یک اشاره‌گر به تابع است. اگر پرانتزها را حذف کنیم، اعلان بصورت زیر در می‌آید:

```
bool *compare( int, int )
```

تابعی اعلان می‌شود که دو آرگومان صحیح بعنوان پارامتر دریافت و یک اشاره‌گر به یک مقدار بولی برگشت خواهد داد.

```
1  // Fig. 8.28: fig08_28.cpp
2  // Multipurpose sorting program using function pointers.
3  #include <iostream>
4  using std::cout;
5  using std::cin;
6  using std::endl;
7
8  #include <iomanip>
9  using std::setw;
10
11 // prototypes
12 void selectionSort( int [], const int, bool (*)( int, int ) );
13 void swap( int * const, int * const );
14 bool ascending( int, int ); // implements ascending order
15 bool descending( int, int ); // implements descending order
16
17 int main()
18 {
19    const int arraySize = 10;
20    int order; // 1 = ascending, 2 = descending
21    int counter; // array index
22    int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
23
24    cout << "Enter 1 to sort in ascending order,\n"
25       << "Enter 2 to sort in descending order: ";
26    cin >> order;
27    cout << "\nData items in original order\n";
28
29    // output original array
30    for ( counter = 0; counter < arraySize; counter++ )
31       cout << setw( 4 ) << a[ counter ];
32
33    // sort array in ascending order; pass function ascending
34    // as an argument to specify ascending sorting order
35    if ( order == 1 )
36    {
37       selectionSort( a, arraySize, ascending );
38       cout << "\nData items in ascending order\n";
39    } // end if
40
41    // sort array in descending order; pass function descending
```

```
42    // as an argument to specify descending sorting order
43    else
44    {
45       selectionSort( a, arraySize, descending );
46       cout << "\nData items in descending order\n";
47    } // end else part of if...else
48
49    // output sorted array
50    for ( counter = 0; counter < arraySize; counter++ )
51       cout << setw( 4 ) << a[ counter ];
52
53    cout << endl;
54    return 0; // indicates successful termination
55 } // end main
56
57 // multipurpose selection sort; the parameter compare is a pointer to
58 // the comparison function that determines the sorting order
59 void selectionSort( int work[], const int size,
60                     bool (*compare)( int, int ) )
61 {
62    int smallestOrLargest; // index of smallest (or largest) element
63
64    // loop over size - 1 elements
65    for ( int i = 0; i < size - 1; i++ )
66    {
67       smallestOrLargest = i; // first index of remaining vector
68
69       // loop to find index of smallest (or largest) element
70       for ( int index = i + 1; index < size; index++ )
71          if ( !(*compare)( work[ smallestOrLargest ], work[ index ] ) )
72             smallestOrLargest = index;
73
74       swap( &work[ smallestOrLargest ], &work[ i ] );
75    } // end if
76 } // end function selectionSort
77
78 // swap values at memory locations to which
79 // element1Ptr and element2Ptr point
80 void swap( int * const element1Ptr, int * const element2Ptr )
81 {
82    int hold = *element1Ptr;
83    *element1Ptr = *element2Ptr;
84    *element2Ptr = hold;
85 } // end function swap
86
87 // determine whether element a is less than
88 // element b for an ascending order sort
89 bool ascending( int a, int b )
90 {
91    return a < b; // returns true if a is less than b
92 } // end function ascending
93
94 // determine whether element a is greater than
95 // element b for a descending order sort
96 bool descending( int a, int b )
97 {
98    return a > b; // returns true if a is greater than b
99 } // end function descending
```

```
Enter1 to sort in ascending order,
Enter 2to sort in descending order: 1

Data item in original order
   2    6    4    8    10    12    89    68    45    37
Data item in ascending order
   2    4    6    8    10    12    37    45    68    89
```

```
Enter1 to sort in ascending order,
Enter 2to sort in descending order: 2
```

```
Data item in original order
    2    6    4    8   10    12    89    68    45    37
Data item in descending order
   89   68   45   37   12    10    8     6     4     2
```

شکل ۲۸-۸ | برنامه مرتب‌سازی با استفاده از اشاره‌گرهای تابع.

پارامتر متناظر در نمونه اولیه تابع **selectionSort** بصورت زیر است

```
bool (*)( int, int)
```

دقت کنید که فقط نوع‌ها در نظر گرفته شده‌اند. مانند همیشه و فقط با هدف مستندسازی، برنامه‌نویس می‌تواند اسامی که توسط کامپایلر نادیده گرفته می‌شوند را هم وارد کند.

تابع ارسالی به **selectionSort** در خط 71 و با عبارت زیر فراخوانی می‌شود:

```
( *compare )( work[ smallestOrLargest ], work[ index ])
```

همانطوری‌که یک اشاره‌گر به یک متغیر ردگیری می‌شود تا به مقدار متغیر دسترسی پیدا شود، یک اشاره‌گر به یک تابع هم ردگیری می‌شود تا تابع را به اجرا در آورد. وجود پرانتزها در اطراف ***compare** ضروری است. اگر این پرانتزها بکار گرفته نشوند، عملگر* مبادرت به ردگیری مقدار برگشتی از فراخوان تابع خواهد کرد.

آرایه‌ای از اشاره‌گرها به توابع

یکی از کاربردهای اشاره‌گر در سیستم‌های متکی بر منو است. برای مثال، برنامه‌ای می‌تواند به کاربر اعلان کند تا گزینه مورد نظر خود را از طریق یک منو با وارد ساختن یک مقدار صحیح انتخاب نماید. از انتخاب کاربر می‌توان بعنوان شاخص در آرایه‌ای از اشاره‌گرهای تابع استفاده کرده و از آن اشاره‌گر برای فراخوانی تابع استفاده کرد.

برنامه شکل ۲۹-۸ یک مثال عادی است که به توصیف اعلان و استفاده از یک آرایه اشاره‌گرها به توابع می‌پردازد. در این برنامه سه تابع بنام‌های **function0**، **function1** و **function2** تعریف شده است که هر یک آرگومان صحیح دریافت و مقداری برگشت نمی‌دهند. خط 17 مبادرت به ذخیره اشاره‌گرهای به این سه تابع در آرایه **f** می‌کند. در این مورد هر سه تابع که آرایه به آنها اشاره می‌کند بایستی دارای نوع برگشتی و پارامترهای یکسان باشند. اگر اعلان بکار رفته در خط 17 را از سمت چپ‌ترین پرانتز بخوانیم به اینصورت خواهد بود «**f** یک آرایه از سه اشاره‌گر به توابعی است که هر یک آرگومانی از نوع **int** دریافت و **void** برگشت می‌دهند.»

```
1  // Fig. 8.29: fig08_29.cpp
2  // Demonstrating an array of pointers to functions.
3  #include <iostream>
4  using std::cout;
5  using std::cin;
6  using std::endl;
7
8  // function prototypes -- each function performs similar actions
9  void function0( int );
10 void function1( int );
11 void function2( int );
```

```
12
13 int main()
14 {
15    // initialize array of 3 pointers to functions that each
16    // take an int argument and return void
17    void (*f[ 3 ])( int ) = { function0, function1, function2 };
18
19    int choice;
20
21    cout << "Enter a number between 0 and 2, 3 to end: ";
22    cin >> choice;
23
24    // process user's choice
25    while ( ( choice >= 0 ) && ( choice < 3 ) )
26    {
27       // invoke the function at location choice in
28       // the array f and pass choice as an argument
29       (*f[ choice ])( choice );
30
31       cout << "Enter a number between 0 and 2, 3 to end: ";
32       cin >> choice;
33    } // end while
34
35    cout << "Program execution completed." << endl;
36    return 0; // indicates successful termination
37 } // end main
38
39 void function0( int a )
40 {
41    cout << "You entered " << a << " so function0 was called\n\n";
42 } // end function function0
43
44 void function1( int b )
45 {
46    cout << "You entered " << b << " so function1 was called\n\n";
47 } // end function function1
48
49 void function2( int c )
50 {
51    cout << "You entered " << c << " so function2 was called\n\n";
52 } // end function function2
```

```
Enter a number between 0 and 2, 3 to end: 0
You entered 0 so function0 was called

Enter a number between 0 and 2, 3 to end: 1
You entered 1 so function1 was called

Enter a number between 0 and 2, 3 to end: 2
You entered 2 so function2 was called

Enter a number between 0 and 2, 3 to end: 3
Program execution completed.
```

شکل ۲۹-۸ | آرایه‌ای از اشاره‌گرها به توابع.

آرایه با اسامی سه تابع (که اشاره‌گر هستند) مقداردهی اولیه شده است. برنامه به کاربر اعلان می‌کند تا عددی مابین 0 و 2 یا برای خاتمه برنامه عدد 3 را وارد سازد. زمانیکه کاربر مقداری مابین 0 و 2 وارد سازد، از این مقدار بعنوان شاخص در آرایه‌ای از اشاره‌گرها به توابع استفاده خواهد شد. خط 29 یکی از توابع موجود در آرایه **f** را فعال می‌سازد. در فرآیند فراخوانی **f[choice]** اشاره‌گری در مکان **choice** آرایه را انتخاب می‌کند. اشاره‌گر برای فراخوانی تابع ردگیری شده و **choice** بعنوان آرگومان به تابع

ارسال می‌گردد. هر تابع مبادرت به چاپ مقدار آرگومان و نام خود می‌کند تا نشان دهد که تابع بدرستی فراخوانی شده است.

۸-۱۳ پردازش رشته‌های مبتنی بر اشاره‌گر

در این بخش، به توضیح برخی از توابع کتابخانه استاندارد C++ که در ارتباط با پردازش رشته هستند، می‌پردازیم. تکنیک‌های مطرح شده در این بخش مناسب ایجاد برنامه‌های ویرایشگر متن، لغت، نرم‌افزار صفحه‌بندی، کامپیوتری کردن سیستم‌های تایپ و انواع نرم‌افزارهای مرتبط با پردازش متن می‌باشد. در حال حاضر از کلاس **string** کتابخانه استاندارد C++ در چندین مثال استفاده کرده‌ایم. برای نمونه، کلاس **GradeBook** در مبحث آموزشی فصل‌های ۳ الی ۷ با استفاده از شی **string** نام دوره‌ای را عرضه می‌کردند. در فصل هیجدهم با جزئیات کلاس **string** آشنا خواهید شد. اگرچه استفاده از شی‌های **string** معمولاً سرراست است، در این بخش از رشته‌های مبتنی بر اشاره‌گر و خاتمه یافته با **null** استفاده خواهیم کرد. تعدادی از توابع کتابخانه استاندارد C++ فقط با رشته‌های خاتمه یافته با **null** و مبتنی بر اشاره‌گر عمل می‌کنند که به نسبت شی‌های **string** پیچیده‌تر هستند. همچنین اگر با برنامه‌های قدیمی C++ کار می‌کنید، احتمالاً نیاز به دستکاری کردن این رشته‌های مبتنی بر اشاره‌گر خواهید داشت.

۸-۱۳-۱ اصول کاراکترها و رشته‌های مبتنی بر اشاره‌گر

کاراکترها بلوک‌های سازنده بنیادین در برنامه‌های منبع C++ هستند. هر برنامه‌ای متشکل از دنباله‌ای از کاراکترها است که وقتی در کنار هم قرار داده می‌شوند مفهوم و معنی پیدا می کنند و می‌توانند توسط کامپایلر بعنوان مجموعه‌ای از دستورات برای انجام وظیفه‌ای، تفسیر گردند. یک برنامه می‌تواند حاوی کاراکترهای ثابت باشد. یک کاراکتر ثابت یک مقدار صحیح است که بعنوان یک کاراکتر در میان علامت نقل قول جای می‌گیرد. مقدار یک کاراکتر ثابت یک مقدار صحیح از کاراکتر در مجموعه کاراکتری کامپیوتر می‌باشد. برای مثال، '2' نشاندهنده مقدار صحیح 2 (عدد 122 در مجموعه کاراکتری ASCII) و '\n' نشاندهنده مقدار خط جدید (عدد 10 در مجموعه کاراکتری ASCII) است.

یک رشته دنباله‌ای از کاراکترها است که با آنها همانند یک یونیت واحد رفتار می‌شود. رشته می‌تواند حاوی حروف، ارقام و کاراکترهای خاص همانند + ، - ، * ، / و $ باشد. رشته‌های لیترال یا رشته‌های ثابت در C++ در میان جفت گوتیشن نوشته می‌شوند، همانند

```
"John Q.Doe"  (نام)
"999 Main Street"  (آدرس خیابان)
"Maynard, Massachusetts" (شهر و ایالت)
"(201) 555-1212"  (شماره تلفن)
```

یک رشته مبتنی بر اشاره‌گر در C++ آرایه‌ای از کاراکترها است که با کاراکتر null خاتمه می‌پذیرند ('**\n**') ، که انتهای رشته در حافظه را مشخص می‌سازد. در دسترسی به یک رشته توسط اشاره‌گر، به اولین

کاراکتر آن دست یافته می‌شود. مقدار یک رشته، آدرس اولین کاراکتر آن است. از اینرو در C++، می‌توان گفت که یک رشته یک اشاره‌گر ثابت است، در واقع یک اشاره‌گر به اولین کاراکتر رشته می‌باشد. در چنین حالتی، رشته‌ها همانند آرایه‌ها هستند چرا که نام آرایه هم یک اشاره‌گر به اولین عنصر آن است.

می‌توان از یک رشته لیترال بعنوان یک مقداردهی کننده در اعلان یک آرایه کاراکتری یا متغیری از نوع **char *** استفاده کرده در اعلان‌های

```
char color[] = "blue";
const char *colorPtr = "blue";
```

هر کدامیک اقدام به مقداردهی یک متغیر با رشته"blue" می‌کند. در اعلان اول، یک آرایه پنج عنصری بنام **color** حاوی کاراکترهای 'e'، 'a'، 'l'، 'b' و'\n' ایجاد می‌شود. اعلان دوم یک متغیر اشاره‌گر بنام **colorPtr** ایجاد می‌کند که به حرف b از رشته "blue" در جائی از حافظه اشاره دارد (که با '**\n**' خاتمه می‌یابد). رشته‌های لیترال دارای کلاس ذخیره سازی **static** هستند (در مدت زمان اجرای برنامه وجود خواهند داشت) و اگر همان رشته لیترال از مکان‌های مختلف برنامه مورد مراجعه قرار گیرد هم می‌تواند و هم نمی‌تواند به اشتراک گذاشته شود. همچنین رشته‌های لیترال، در C++ ثابت هستند و کاراکترهای آنها قابل تغییر نمی‌باشد.

اعلان **char color[ ] = "blue";** می‌تواند بصورت زیر هم نوشته شود

```
char color[] = {'b', 'l', 'u', 'e', '\n'};
```

در زمان اعلان یک آرایه کاراکتری که حاوی رشته خواهد بود، بایستی آرایه به میزان کافی برای ذخیره سازی رشته و کاراکتر null آن، بزرگ باشد. در اعلان فوق سایز آرایه براساس تعداد موجود در لیست مقداردهی اولیه مشخص است.

خطای برنامه‌نویسی

*عدم تخصیص فضای کافی در یک آرایه کاراکتری برای ذخیره کاراکتر **null** که پایان دهنده یک رشته است، خطا خواهد بود.*

خطای برنامه‌نویسی

*ایجاد یا استفاده از یک رشته مبتنی بر C یا C-style که حاوی کاراکتر **null** نمی‌باشد، می‌تواند خطاهای منطقی بدنبال داشته باشد.*

اجتناب از خطا

در زمان ذخیره سازی رشته‌ای از کاراکترها در یک آرایه کاراکتری، مطمئن شوید که آرایه به میزان کافی فضا برای نگهداری بزرگترین رشته‌ای که در آن ذخیره خواهد شد، در اختیار دارد. C++ به رشته‌ها به هر طولی اجازه ذخیره شدن می‌دهد. اگر رشته‌ای طولانی‌تر از آرایه کاراکتری باشد که در آن ذخیره خواهد شد،

کاراکترهای قرار گرفته در آن سوی مرز آرایه بر روی داده‌های موجود در حافظه پس از آرایه بازنویسی می‌شوند و این عمل می‌تواند خطاهای منطقی بوجود آورد.

یک رشته می‌تواند بدون یک آرایه کاراکتری با استفاده از **cin** خوانده شود. برای مثال، از عبارت زیر می‌توان برای خواندن یک رشته بدرون آرایه **word[20]** استفاده کرد:

```
cin >> word;
```

رشته وارد شده توسط کاربر در **word** ذخیره می‌شود. عبارت فوق مبادرت به خواندن کاراکترها تا رسیدن به یک کاراکتر فاصله یا شاخص انتهای فایل می‌کند. توجه کنید که رشته نبایستی بیش از 19 کاراکتر طول داشته باشد تا مکانی هم برای کاراکتر null بکار گرفته شود، از دستور **setw** می‌توان برای مطمئن شدن از اینکه رشته خوانده شده به **word** از سایز آرایه تجاوز نکرده استفاده کرد. برای مثال، عبارت

```
cin >> setw(20) >> word;
```

مشخص می‌کند که **cin** بایستی حداکثر 19 کاراکتر بدرون آرایه **word** بخواند و در مکان بیستم آرایه، کاراکتر خاتمه‌دهنده null را برای رشته ذخیره نماید. دستور **setw** فقط پس از ورودی مقدار عمل می‌کند. اگر بیش از 19 کاراکتر وارد شده باشد، مابقی کاراکترها در آرایه **word** ذخیره نمی‌شوند، اما خوانده شده و می‌توانند در متغیر دیگری ذخیره گردند.

در برخی از مواقع مناسب خواهد بود تا کل یک خط بعنوان ورودی در آرایه وارد گردد. به همین منظور C++ تابع **cin.getline** در سرآیند فایل <**iostream**> را در نظر گرفته است. در فصل سوم به معرفی تابع مشابهی بنام **getline** از سرآیند فایل <**string**> پرداختیم، که ورودی را تا مواجه شدن با یک کاراکتر خط جدید خوانده و آنرا در یک رشته مشخص شده بعنوان آرگومان ذخیره می‌سازد (البته بدون کاراکتر خط جدید). تابع **cin.getline** سه آرگومان دریافت می‌کند، یک آرایه کاراکتری که خطی از متن در آن ذخیره می‌شود، طول و کاراکتر نشاندهنده انتهای داده یا حائل. برای مثال، در بخشی از برنامه زیر

```
char sentence[80];
cin.getline( sentence, 80, '\n' );
```

آرایه **sentence** هشتاد کاراکتری را اعلان و یک خط از متن را از صفحه کلید بدرون آرایه می‌خواند. تابع تا رسیدن به کاراکتر حائل '**\n**' به خواندن کاراکترها ادامه می‌دهد و زمانیکه تعیین کننده انتهای فایل وارد شد یا تعداد کاراکترها خوانده شده یک‌ی بیش از طول تعیین شده در آرگومان دوم گردید، خواندن متوقف می‌شود (کاراکتر آخر در آرایه برای کاراکتر null رزرو شده است). اگر با کاراکتر حائل برخورد کند آنرا خوانده و حذف می‌نماید. آرگومان سوم در **cin.getline** آرگومان '**\n**' است که مقدار پیش‌فرض است، از اینرو فراخوانی تابع فوق می‌توانست بصورت زیر نوشته شود:

```
cin.getline( sentence, 80 );
```

در فصل پانزدهم با جزئیات عملکرد **cin.getline** و دیگر توابع ورودی/خروجی آشنا خواهید شد.

خطای برنامه‌نویسی

ارسال یک رشته بعنوان آرگومان به تابعی که انتظار یک کاراکتر را دارد، خطای کامپایل است.

۸-۱۳-۲ توابع دستکاری کننده رشته

توابع مرتبط با رشته در کتابخانه توابع، توابع مناسبی برای دستکاری داده‌های رشته‌ای، مقایسه رشته‌ها، جستجو رشته‌ها برای یافتن کاراکترهای خاص و دیگر رشته‌ها، نشانه‌گذاری رشته‌ها (مجزا کردن رشته‌ها به قسمتهای منطقی بعنوان کلمات مجزا در یک جمله) و تعیین طول رشته‌ها، هستند. در این بخش به معرفی تعدادی از توابع پرکاربرد در زمینه دستکاری رشته‌ها از کتابخانه استاندارد C++ خواهیم پرداخت. در جدول شکل ۳۰-۸ این توابع آورده شده‌اند و از آنها در مثال‌هایی استفاده کرده‌ایم. نمونه اولیه این توابع در فایل سرآیند <**cstring**> قرار دارند.

نمونه اولیه تابع	توضیح
`char *strcpy( char *S1, const char *S2 );`	رشته s2 را بدرون آرایه کاراکتری s1 کپی می‌کند. مقدار s1 برگشت داده می‌شود.
`char *strncpy( char *s1, const char *s2, size_t n);`	حداکثر n کاراکتر از رشته s2 را بدرون آرایه کارکتری s1 کپی می‌کند. مقدار s1 برگشت داده می‌شود.
`char *strcat ( char *s1, const char *s2 )`	رشته s2 را به s1 الصاق می‌کند. اولین کاراکتر s2 مبادرت به بازنویسی کاراکتر null از s1 می‌نماید. مقدار s1 برگشت داده می‌شود.
`char *strncat( char *s1, const char *s2, size_t n);`	حداکثر n کاراکتر از رشته s2 را به s1 الصاق می‌کند. اولین کاراکتر s2 مبادرت به بازنویسی کاراکتر null از s1 می‌نماید. مقدار s1 برگشت داده می‌شود.
`int strcmp( const char *s1, const char *s2 );`	رشته s1 را با رشته s2 مقایسه می‌کند. تابع مقدار صفر، کمتر از صفر (معمولاً 1-) یا بزرگتر از صفر (معمولاً 1) در صورتیکه به ترتیب s1 برابر، کوچکتر یا بزرگتر از s2 باشد، برگشت می‌دهد.
`int strncmp( const char *s1, const char *s2, size_t n );`	تا n کاراکتر مبادرت به مقایسه رشته s1 با رشته s2 می‌کند. تابع مقدار صفر، کمتر از صفر یا بزرگتر از صفر برگشت می‌دهد اگر بخش n کاراکتری از s1 برابر، کمتر یا بزرگتر از n کاراکتر متناظر با رشته s2 باشد (به‌ترتیب).
`char *strtok( char *s1, const char *s2 );`	با فراخوانی دنباله‌ای از strtok، رشته s1 به نشانه یا توکن‌های تقسیم می‌شود. تقسیم رشته برپایه کاراکترهای موجود در رشته s2 است. برای نمونه، اگر رشته "this: is: a: string" را برپایه کاراکتر ':' تقسیم کنیم، نتیجه نشانه‌گذاری "a" ، "is" ، "this" و "string" خواهد بود. تابع strtok در هر بار یک نشانه برگشت می‌دهد. اولین فراخوانی حاوی s1 بعنوان اولین آرگومان و فراخوانی‌ها متعاقب آن مبادرت به نشانه‌گذاری همان رشته حاوی NULL بعنوان اولین آرگومان ادامه خواهد داد. زمانیکه دیگر چیزی باقی نمانده باشد، تابع NULL برگشت خواهد داد.
`size_t strlen( const char *s );`	تعیین کننده طول رشته s است.

شکل ۳۰-۸ | توابع دستکاری کننده رشته از کتابخانه استاندارد رشته.

کپی رشته‌ها با strcpy و strncpy

تابع **strcpy** مبادرت به کپی دومین آرگومان خود (رشته) بدرون اولین آرگومان خود می‌کند که یک آرایه کاراکتری است که باید به میزان کافی برای ذخیره‌سازی رشته و کاراکتر null بزرگ باشد. تابع **strncpy** عملکردی همانند **strcpy** دارد، بجز اینکه این تابع به تعداد مشخص کاراکتر را از رشته به آرایه کپی می‌کند. توجه کنید که تابع **strncpy** ضرورتاً مبادرت به کپی کاراکتر خاتمه‌دهنده null در آرگومان دوم خود نمی‌کند، فقط در صورتی کاراکتر null کپی خواهد شد که تعداد کاراکترهای کپی شونده حداقل یک کاراکتر بیش از رشته طول داشته باشند. برای مثال، اگر "test" آرگومان دوم باشد، کاراکتر null فقط در صورتیکه آرگومان سوم در **strcnpy** حداقل 5 باشد کپی خواهد شد (چهار کاراکتر در "test" به اضافه یک کاراکتر null). اگر آرگومان سوم بزرگتر از 5 باشد، کاراکترهای null به آرایه الصاق می‌شوند تا اینکه مجموع تعداد کاراکترها مشخص شده توسط آرگومان سوم نوشته شود.

در برنامه شکل ۳۱-۸ از تابع **strcpy** در خط 17 برای کپی کل رشته موجود در آرایه **x** بدرون آرایه **y** و از تابع **strncpy** (خط 23) برای کپی 14 کاراکتر اول از آرایه **x** به آرایه **z** استفاده شده است. خط 24 مبادرت به الصاق کاراکتر null ('**\n**') به آرایه **z** می‌کند، چرا که فراخوانی **strncpy** در برنامه مبادرت به نوشتن کاراکتر null نمی‌کند (آرگومان سوم کمتر از طول آرگومان دوم به اضافه یک است).

```
1  // Fig. 8.31: fig08_31.cpp
2  // Using strcpy and strncpy.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <cstring> // prototypes for strcpy and strncpy
8  using std::strcpy;
9  using std::strncpy;
10
11 int main()
12 {
13    char x[] = "Happy Birthday to You"; // string length 21
14    char y[ 25 ];
15    char z[ 15 ];
16
17    strcpy( y, x ); // copy contents of x into y
18
19    cout << "The string in array x is: " << x
20       << "\nThe string in array y is: " << y << '\n';
21
22    // copy first 14 characters of x into z
23    strncpy( z, x, 14 ); // does not copy null character
24    z[ 14 ] = '\0'; // append '\0' to z's contents
25
26    cout << "The string in array z is: " << z << endl;
27    return 0; // indicates successful termination
28 } // end main
```

```
The string in array x is: Happy Birthday to You
The string in array y is: Happy Birthday to You
The string in array z is: Happy Birthday
```

شکل ۳۱-۸ | توابع strcpy و strncpy.

الصاق رشته‌ها با strcat و strncat

تابع **strcat** مبادرت به الصاق آرگومان دوم خود (یک رشته) به آرگومان اول خود (یک آرایه کاراکتری حاوی رشته) می‌کند. کاراکتر اول از آرگومان دوم جایگزین کاراکتر null می‌شود ('\n') که خاتمه‌دهنده رشته در آرگومان اول است. برنامه‌نویس باید مطمئن شود که آرایه بکار رفته برای ذخیره رشته اول به اندازه کافی برای ذخیره‌سازی ترکیب رشته اول، رشته دوم و کاراکتر null بزرگ است. تابع **strncat** مبادرت به الصاق تعداد مشخصی از کاراکترها از رشته دوم به رشته اول کرده و یک کاراکتر null به نتیجه اضافه می‌کند. برنامه شکل ۲۴–۸ به توصیف عملکرد توابع **strcat** (خطوط 19 و 29) و **strncat** (خط 24) می‌پردازد.

```
1  // Fig. 8.32: fig08_32.cpp
2  // Using strcat and strncat.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <cstring> // prototypes for strcat and strncat
8  using std::strcat;
9  using std::strncat;
10
11 int main()
12 {
13    char s1[ 20 ] = "Happy "; // length 6
14    char s2[] = "New Year "; // length 9
15    char s3[ 40 ] = "";
16
17    cout << "s1 = " << s1 << "\ns2 = " << s2;
18
19    strcat( s1, s2 ); // concatenate s2 to s1 (length 15)
20
21    cout << "\n\nAfter strcat(s1, s2):\ns1 = " << s1 << "\ns2 = " << s2;
22
23    // concatenate first 6 characters of s1 to s3
24    strncat( s3, s1, 6 ); // places '\0' after last character
25
26    cout << "\n\nAfter strncat(s3, s1, 6):\ns1 = " << s1
27       << "\ns3 = " << s3;
28
29    strcat( s3, s1 ); // concatenate s1 to s3
30    cout << "\n\nAfter strcat(s3, s1):\ns1 = " << s1
31       << "\ns3 = " << s3 << endl;
32    return 0; // indicates successful termination
33 } // end main
```

```
s1 = Happy
s2 = New Year

After strcat(s1, s2):
s1 = Happy New Year
s2 = New Year

After strncat(s3, s1, 6):
s1 = Happy New Year
s3 = Happy

After strcat(s3, s1):
s1 = Happy New Year
s3 = Happy Happy New Year
```

شکل ۳۲–۸ | توابع strcat و strncat.

***مقایسه رشته‌ها با* strcmp *و* strncmp**

در برنامه شکل ۳۳-۸ سه رشته با استفاده از توابع **strcmp** (در خطوط 21، 22 و 23) و **strncmp** (در خطوط 26، 27 و 28) مقایسه شده‌اند. تابع **strcmp** مبادرت به مقایسه کاراکتر به کاراکتر اولین آرگومان رشته‌ای با دومین آرگومان رشته‌ای خود می‌کند. در صورتیکه رشته‌ها باهم برابر باشند تابع مقدار صفر، و در صورتیکه رشته اول کوچکتر از رشته دوم باشد مقدار منفی و اگر رشته اول بزرگتر از رشته دوم باشد مقدار مثبت برگشت می‌دهد. عملکرد تابع **strncmp** همانند **strcmp** است بجز اینکه **strncmp** به تعداد مشخص از کاراکترها شروع به مقایسه می‌کند. تابع **strncmp** در صورتیکه به کاراکتر null در یکی از آرگومان‌های رشته‌ای خود برسد، عملیات مقایسه را متوقف خواهد کرد. برنامه در هر بار فراخوانی تابع یک مقدار صحیح برگشت می‌دهد.

```
1  // Fig. 8.33: fig08_33.cpp
2  // Using strcmp and strncmp.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <iomanip>
8  using std::setw;
9
10 #include <cstring> // prototypes for strcmp and strncmp
11 using std::strcmp;
12 using std::strncmp;
13
14 int main()
15 {
16    char *s1 = "Happy New Year";
17    char *s2 = "Happy New Year";
18    char *s3 = "Happy Holidays";
19
20    cout << "s1 = " << s1 << "\ns2 = " << s2 << "\ns3 = " << s3
21       << "\n\nstrcmp(s1, s2) = " << setw( 2 ) << strcmp( s1, s2 )
22       << "\nstrcmp(s1, s3) = " << setw( 2 ) << strcmp( s1, s3 )
23       << "\nstrcmp(s3, s1) = " << setw( 2 ) << strcmp( s3, s1 );
24
25    cout << "\n\nstrncmp(s1, s3, 6) = " << setw( 2 )
26       << strncmp( s1, s3, 6 ) << "\nstrncmp(s1, s3, 7) = " << setw( 2 )
27       << strncmp( s1, s3, 7 ) << "\nstrncmp(s3, s1, 7) = " << setw( 2 )
28       << strncmp( s3, s1, 7 ) << endl;
29    return 0; // indicates successful termination
30 } // end main
```

```
s1 = Happy New Year
s2 = Happy New Year
s3 = Happy Holidays

strcmp(s1,s2) =  0
strcmp(s1,s3) =  1
strcmp(s3,s1) = -1

strcmd(s1, s3, 6) =  0
strcmd(s1, s3, 7) =  1
strcmd(s3, s1, 7) = -1
```

شکل ۳۳-۸ | توابع strcmp **و** strncmp.

برای اینکه بخوبی با مفهوم جمله یک رشته «بزرگتر» یا «کوچکتر» از رشته دیگر است آشنا شوید، به فرآیند به ترتیب الفباء نوشتن نام‌های خانوادگی دقت کنید. نبایستی در قرار دادن"Jones" قبل از "Smith" شک کنید چرا که حرف اول"Jones" قبل از حرف اول "Smith" در الفباء آمده است. اما الفباء چیزی بیش از لیست 26 حرفی است و یک لیست مرتب شده از کاراکترها می‌باشد. هر حرف در مکان خاصی از لیست جای دارد.

چگونه کامپیوتر از قرار گرفتن کاراکتری قبل از کاراکتر دیگری مطلع است؟ تمام کاراکترهای عرضه شده در کامپیوتر دارای کدهای عددی هستند و زمانیکه کامپیوتر مبادرت به مقایسه دو رشته می‌کند، در واقع به مقایسه کدهای عددی کاراکترهای موجود در رشته می‌پردازد.

برای استانداردسازی عرضه کاراکترها، اکثر سازندگان کامپیوتر، براساس یکی از دو طرح کدگذاری، *ASCII* یا *EBCODIC* کامپیوترهای خود را می‌سازند. بخاطر دارید که ASCII سرنام کلمات " *American Standard Code for Information Interchang*" و EBCDC سرنام کلمات " Extended *Binary Coded Decimal Interchange Code*" است. البته طرح‌های کدگذاری دیگری هم وجود دارند اما این دو طرح از محبوبیت بیشتری برخوردار هستند. به ASCII و EBCDIC *کدهای کاراکتر* یا *مجموعه کاراکتری* هم گفته می‌شود.

نشانه‌گذاری رشته با strtok

تابع **strtok** مبادرت به تقسیم یک رشته به دنباله‌ای از *نشانه‌ها* یا *توکن‌ها* می‌کند. یک توکن توالی از کاراکترهای مجزا شده توسط کاراکترهای حائل است (معمولاً فاصله یا نماد نقطه‌گذاری). برای مثال، در یک خط از متن هر کلمه می‌تواند بعنوان یک توکن در نظر گرفته شود و فاصله بین کلمات می‌توانند بعنوان حائل مورد توجه واقع شوند.

برای تقسیم یک رشته‌ها به توکن‌ها، نیاز به فراخوانی‌های مضاعف **strtok** است(با فرض اینکه رشته حاوی بیش از یک توکن است). اولین فراخوانی **strtok** حاوی دو آرگومان، یکی رشته برای توکن شدن و دیگری رشته‌ای حاوی حائل است. در خط 19 برنامه شکل ۳۴-۸ مبادرت به اشاره دادن **tokenPtr** به اولین توکن در **sentence** شده است. آرگومان دوم،" "، نشان می‌دهد که توکن‌ها در **sentence** توسط فاصله از هم متمایز خواهند شد. تابع **strtok** جستجویی برای یافتن اولین کاراکتر در **sentence** می‌کند که یک کاراکتر حائل نیست (فاصله). این شروع اولین توکن است. سپس تابع، کاراکتر حائل بعدی در رشته را پیدا کرده و آنرا با یک کاراکتر null جایگزین می‌سازد. ('**n**'). در اینحالت توکن جاری خاتمه می‌پذیرد. تابع **strtok** اشاره‌گر را به کاراکتری بعدی پس از توکن در **sentence** انتقال می‌دهد و یک اشاره‌گر به توکن جاری برگشت داده می‌شود.

```
1  // Fig. 8.34: fig08_34.cpp
2  // Using strtok.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <cstring> // prototype for strtok
8  using std::strtok;
9
10 int main()
11 {
12    char sentence[] = "This is a sentence with 7 tokens";
13    char *tokenPtr;
14
15    cout << "The string to be tokenized is:\n" << sentence
16       << "\n\nThe tokens are:\n\n";
17
18    // begin tokenization of sentence
19    tokenPtr = strtok( sentence, " " );
20
21    // continue tokenizing sentence until tokenPtr becomes NULL
22    while ( tokenPtr != NULL )
23    {
24       cout << tokenPtr << '\n';
25       tokenPtr = strtok( NULL, " " ); // get next token
26    } // end while
27
28    cout << "\nAfter strtok, sentence = " << sentence << endl;
29    return 0; // indicates successful termination
30 } // end main
```

```
The string to be tokenized is :
This is a sentence with 7 tokens

The tokens are:

This
is
a
sentence
with
7
tokens

After strtok, sentence = This
```

شکل ۳٤–۸ | تابع strtok.

با فراخوانی‌های پشت سرهم **strtok**، رشته sentence توکن می‌شود که **NULL** بعنوان آرگومان اول ارسال می‌گردد. آرگومان NULL بر این نکته تاکید دارد که فراخوانی **strtok** بایستی عملیات توکن کردن را از مکانی در sentence انجام دهد که در آخرین فراخوانی **strtok** باقی مانده است. توجه کنید که **strtok** این اطلاعات را به روشی ذخیره می‌سازد که در دید برنامه‌نویس قرار ندارند. اگر هیچ توکنی به هنگام فراخوانی **strtok** باقی نمانده باشد، تابع مقدار **NULL** برگشت می‌دهد. برنامه شکل ۳۴–۸ با استفاده از تابع **strtok** مبادرت به توکن کردن رشته "This is a sentence with 7 tokens" کرده است. برنامه هر توکن را در یک خط متمایز چاپ می‌کند. خط 28 پس از توکن‌سازی جمله مبادرت به چاپ sentence می‌کند. توجه کنید که تابع **strtok** در رشته ورودی تغییر بوجود می‌آورد، از اینرو، یکی کپی از رشته در صورتیکه برنامه نیاز به رشته اصلی پس از فراخوانی **strtok** داشته باشد، تهیه کنید. مشاهده

می‌کنید که پس از توکن کردن رشته، زمانیکه sentence چاپ می‌شود فقط حاوی کلمه "This" است، چرا که **strtok** هر فاصله در sentence را با یک کاراکتر null در زمان توکن کردن جایگزین ساخته است.

تعیین طول رشته

تابع **strlen** یک رشته بعنوان یک آرگومان دریافت و تعداد کاراکترهای موجود در رشته را برگشت می‌دهد (کاراکتر null در طول رشته محاسبه نمی‌شود). برنامه شکل ۳۵-۸ به بررسی عملکرد تابع **strlen** پرداخته است.

```
1  // Fig. 8.35: fig08_35.cpp
2  // Using strlen.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <cstring> // prototype for strlen
8  using std::strlen;
9
10 int main()
11 {
12    char *string1 = "abcdefghijklmnopqrstuvwxyz";
13    char *string2 = "four";
14    char *string3 = "Boston";
15
16    cout << "The length of \"" << string1<< "\" is "<< strlen( string1 )
17       << "\nThe length of \"" << string2<< "\" is "<< strlen( string2 )
18       << "\nThe length of \"" << string3<< "\" is "<< strlen( string3 )
19       << endl;
20    return 0; // indicates successful termination
21 } // end main
```

```
The length of "abcdefghijklmnopqrstuvwxyz" is 26
The length of "four" is 4
The length of "Boston" is 6
```

شکل ۳۵-۸ | تابع strlen طول یک رشته *char را برگشت می‌دهد.

خودآزمایی

۱-۸ به موارد زیر پاسخ دهید:

a) اشاره‌گر متغیری است که مقدار آن ..... یک متغیر دیگر است.

b) سه مقداری که می‌توان در مقداردهی اولیه یک اشاره‌گر بکار گرفت عبارتند از .....، ..... و .....

c) تنها مقدار صحیحی که می‌توان مستقیماً به یک اشاره‌گر تخصیص داد، مقدار..... است.

۲-۸ کدامیک از موارد زیر صحیح و کدامیک اشتباه است.

a) عملگر آدرس & می‌تواند فقط بر روی ثابت‌ها و عبارات بکار گرفته شود.

b) اشاره‌گری که از نوع void * اعلان شده است می‌تواند ردگیری شود.

c) اشاره‌گرها از انواع مختلف هرگز نمی‌توانند بدون عملیات تبدیل (cast) به نوع دیگری تخصیص داده شوند.

۸–۳ برای هر یک از موارد زیر عبارتی در C++ بنویسید که وظیفه درخواستی را به انجام برساند. فرض کنید که اعداد با دقت مضاعف، اعشاری در هشت بایت ذخیره می‌شوند و آدرس شروع آرایه در مکان 1002500 قرار دارد. هر بخش از تمرین باید از نتایج بخش‌های قبلی استفاده کند.

a) آرایه‌ای از نوع double بنام numbers با 10 عنصر اعلان کرده و عناصر آرایه را با مقادیر 0.0,1.1,2.2,...,9.9 مقداردهی کنید. فرض کنید که ثابت نمادین SIZE با مقدار 10 تعریف شده باشد.

b) اشاره‌گر nPtr را اعلان کنید که به متغیری از نوع double اشاره کند.

c) از یک عبارت for برای چاپ عناصر آرایه numbers با استفاده از شاخص آرایه استفاده کنید. هر عدد را با دقت یک رقم در سمت راست نقطه دیسمال چاپ کنید.

d) دو عبارت مجزا بنویسید که به هر یک آدرس شروع آرایه numbers را به متغیر اشاره‌گر nPtr تخصیص دهد.

e) با استفاده از عبارت for عناصر آرایه numbers را با استفاده از نماد اشاره‌گر/افست با اشاره‌گر nPtr چاپ کنید.

f) با استفاده از عبارت for عناصر آرایه numbers را با استفاده از نماد اشاره‌گر/افست و توسط نام آرایه بعنوان اشاره‌گر چاپ کنید.

g) با استفاده از عبارت for عناصر آرایه numbers را با استفاده از نماد اشاره‌گر/افست با اشاره‌گر nPtr چاپ کنید.

h) به پنجمین عنصر آرایه numbers با استفاده از شاخص اشاره‌گر/افست با نام آرایه بعنوان اشاره‌گر، نماد شاخص اشاره‌گر با nPtr و نماد اشاره‌گر/افست با nPtr مراجعه کنید.

i) فرض کنید که nPtr به ابتدای آرایه numbers اشاره می‌کند، آدرس مورد مراجعه، کدام آدرس توسط nPrt + 8 مورد مراجعه قرار می‌گیرد؟ چه مقداری در آن مکان ذخیره شده است؟

j) فرض کنید که nPtr به numbers[5] اشاره دارد، کدام آدرس توسط nPtr پس از nPtr -= 4 بکار گرفته خواهد شد؟ چه مقداری در آن مکان ذخیره شده است؟

۸–۴ برای هر یک از موارد زیر، یک عبارت بنویسید که کار خواسته شده را انجام دهد. فرض کنید که متغیرهای اعشاری number1 و number2 اعلان شده‌اند و number1 با 7.3 مقداردهی اولیه شده است. فرض کنید که متغیر ptr از نوع char * است. همچنین فرض نمائید که آرایه‌های s1 , s2 هر یک آرایه‌های 100 عنصری از نوع char هستند که با رشته لیترال مقداردهی اولیه شده‌اند.

a) اعلان متغیر fPtr برای اشاره به یک شی از نوع double.

b) تخصیص آدرس متغیر number1 برای اشاره به متغیر fPtr.

c) چاپ مقدار شی مورد اشاره توسط fPtr.

d) تخصیص مقدار شی مورد اشاره توسط fPtr به متغیر number2.

e) چاپ مقدار number2.

f) چاپ آدرس number1.

g) چاپ آدرس ذخیره شده در fPtr. آیا مقدار چاپ شده همان آدرس number1 است؟

h) کپی رشته ذخیره شده در آرایه s2 به آرایه s1.

i) مقایسه رشته موجود در s1 با رشته s2 و چاپ نتیجه.

j) اضافه کردن 10 کاراکتر اول از رشته s2 به رشته s1.

k) تعیین طول رشته موجود در s1 و چاپ نتیجه.

L) تخصیص مکان اولین توکن در s2 به ptr.توکن‌ها توسط کاما (،) از هم مجزا می‌شوند.

۸-۵ برای برآورده کردن هر یک از موارد زیر، عبارتی بنویسید:

a) سرآیندی برای تابعی بنام exchange بنویسید که دو اشاره‌گر به مقدار اعشاری با دقت مضاعف x , y دریافت کرده و مقداری برگشت نمی‌دهد.

b) نمونه اولیه تابع برای تابع معرفی شده در بخش (a) بنویسید.

c) سرآیندی برای تابع evaluate بنویسید که یک مقدار صحیح برگشت داده و پارامتری x از نوع صحیح و اشاره‌گری به تابع poly را دریافت نماید. تابع poly یک پارامتر صحیح دریافت کرده و مقدار صحیحی را برگشت می‌دهد.

d) نمونه اولیه تابع برای تابع معرفی شده در بخش (c) بنویسید.

e) دو عبارت بنویسید که هر یک آرایه کاراکتری vowel را با رشته "AEIOU" مقداردهی اولیه کنند.

۸-۶ در هر یک از بخش‌های زیر خطای رخ داده را پیدا کنید. فرض کنید که اعلان‌ها و عبارات زیر صورت گرفته‌اند:

```
int *zPtr ; //zPtr will refernce array z
int *aPtr = 0;
voild *sPtr = 0;
int number;
int z[5] = {1,2,3,4,5};

b)
//use pointer to get first value of array
Number=zPtr;
c)
//assign array element 2 (the value 3) to number
number=*zPtr[2];
d)
//print entire array z
for(int i=0; i<=5; i++)
 cout<<zPtr[i]<<endl;
e)
//assign the value pointed to by sPtr to number
number=*sPtr;
f)
++z;
g)
```

```
char s[10];
cout<<strncpy(s,"hello",5)<<endl;
h)
char s[12];
strcpy(s, "Welcome home");
i)
if(strcmp(string1,string2))
cout<<"The strings are equal"<<endl;
```

۸-۷ به هنگام اجرای هر یک از عبارات زیر چه جمله‌ای چاپ خواهد شد؟ اگر عبارتی حاوی خطا باشد، خطا را توضیح داده و نحوه اصلاح آنرا بیان کنید. فرض کنید متغیرهای زیر اعلان شده‌اند:

```
Char s1[50]= "Jack";
Char s2[60]="jill";
Char s3[50];

a) cout<<strcpy(s3,s2)<<endl;
b) cout<<strcat(strcat(strcpy(s3,s1), "and"),s2)<<endl;
c) cout<<strlen(s1) + strlen(s2)<<endl;
d) cout<<strlen(s3)<<endl;
```

پاسخ خودآزمایی

۸-۱

a) آدرس. b) صفر، Null، آدرس. c) صفر.

۸-۲

a) اشتباه. عملوند، عملگر آدرس بایستی یک lvalue باشد. عملگر آدرس نمی‌تواند با ثابت‌ها یا عباراتی بکار رود که نتیجه‌ای از مراجعه‌ها ندارند.

b) اشتباه. اشاره‌گر به void نمی‌تواند ردگیری شود. چنین اشاره‌گری دارای نوع نیست که کامپایلر بتواند تعداد بایت‌های حافظه را برای ردگیری و نوع داده که اشاره‌گر به آن اشاره می‌کند را تعیین کند.

c) اشتباه. اشاره‌گرها از هر نوع می‌توانند به اشاره‌گرهای void تخصیص یابند. اشاره‌گرها از نوع void می‌توانند فقط با تبدیل صریح نوع به اشاره‌گرهایی از نوع دیگر تخصیص داده شوند.

۸-۳

```
a)
double numbers[SIZE]={0.0,1.1,2.2,3.3,4.4,5.5,6.6,7.7,8.8,9.9};
b)
double *nPtr;
c)
cout<<fixed<<showpoint<<setprecision(1);
 for(int i=0; i<SIZE;i++)
   cout<<numbers[i]<<' ';
d)
nPtr=numbers;
nPtr=&numbers[0];
```

e)
```
cout<<fixed<<showpoint<<setprecision(1);
 for(int j=0; j<SIZE;j++)
  cout<<*(nPtr + j)<<' ';
```
f)
```
cout<<fixed<<showpoint<<setprecision(1);
 for(int k=0; k<SIZE;K++)
  cout<< *(numbers +k) <<' ';
```
g)
```
cout<<fixed<<showpoint<<setprecision(1);
 for(int m=0; m<SIZE; m++)
  cout<<nPtr[m]<<' ';
```
h)
```
numbers[3]
*(numbers+3)
nPtr[3]
*(nPtr + 3)
```

i) آدرس برابر است با 1002564=1002500+8*8، مقدار 8.8 است.

j) آدرس numbers[5] برابر است با 1002540=1002500+5*8

آدرس nPtr -=4 برابر است با 1002508=1002540-4*8

مقدار موجود در آن مکان برابر 1.1 است.

۸–۴

```
a) double *fPtr;
b) fPtr=&number1;
c) cout<<"The value of *fPtr is"<<*fPtr<<endl;
d) number2=*fPtr;
e) cout<<"The value of number2 is"<<number2<<endl;
f) cout<<"The address of number1 is"<<&number1<<endl;
g) cout<<"The address stored in fPtr is"<<fPtr is"<<fPtr<<endl;
h) strcpy(s1,s2);
i) cout<<"strcmp(s1,s2)="<<strcmp(s1,s2)<<endl;
j) strncat(s1,s2,10);
k) cout<<"strlen(s1)="<<strlen(s1)<<endl;
l) ptr= strtok(s2,",");
```

۸–۵

```
a) void exchange (double **,double *y)
b) void exchange(double*,double*);
c) int evaluate (int x,int(*poly)(int))
d) int evaluate(int,int(*)(int));
e) char vawe[] = "AEIOU";
   char vawe[] = {'A','E','I','O','U','10'};
```

۸–۶

a) خطا: ptr= مقدار دهی نشده است.

اصلاح: مقداردهی zftr با ;zptr=z

b) خطا: اشاره‌گر ردگیری نشده است.

اصلاح: تغییر عبارت به ;number= *zPtr

c) خطا: zPtr[2] اشاره‌گر نبوده و نمی‌توان ردگیری شود.

اصلاح: تغییر zPtr[2]* به zPtr[2]

e) خطا: مراجعه به عنصر آرایه خارج از مرزهای آرایه با شاخص اشاره‌گر.

اصلاح: برای اجتناب از اینکار, عملکرد رابطه در عبارت for را به < تغییر دهید.

۸-۷

a) jill
b) jack and jill
c) 8
d) 13

۸-۸ مشخص کنید که آیا عبارات زیر صحیح هستند یا اشتباه. اگر اشتباه هستند، علت را توضیح دهید.

a) دو اشاره‌گر که به آرایه‌های متفاوت اشاره می‌کنند، نمی‌توانند بطور موثر با هم مقایسه شوند.

b) چون نام آرایه یک اشاره‌گر به اولین عنصر آرایه است، اسامی آرایه‌ها می‌توانند به همان روش اشاره‌گرها، دچار تغییر شوند.

۸-۹ برای هر یک از موارد زیر، عباراتی به C++ بنویسید که کار درخواستی را انجام دهند. فرض کنید که مقادیر صحیح بدون علامت در دو بایت ذخیره شده و آدرس شروع آرایه در مکان 1002500 قرار دارد.

a) آرایه از نوع unsigned int بنام values با پنج عنصر اعلان کنید. عناصر آرایه را با مقادیر زوج 2 تا 10 مقداردهی اولیه نمائید. فرض کنید نماد ثابت SIZE با 5 تعریف شده است.

b) اعلان اشاره‌گر vPtr که به شی از نوع unsigned int اشاره می‌کند.

c) استفاده از عبارت for برای چاپ عناصر آرایه values با استفاده از شاخص.

d) دو عبارت مجزا بنویسید که آدرس شروع آرایه values را به متغیر اشاره‌گر vPtr تخصیص دهد.

e) استفاده از عبارت for برای چاپ عناصر آرایه values با استفاده از نماد اشاره‌گر/افست.

f) استفاده از عبارت for برای چاپ عناصر آرایه values با استفاده از نماد اشاره‌گر/افست به همراه نام آرایه بعنوان اشاره‌گر.

g) استفاده از عبارت for برای چاپ عناصر آرایه values توسط شاخص.

h) مراجعه به عنصر پنجم values با استفاده از شاخص آرایه، اشاره‌گر/افست توسط نام آرایه بعنوان اشاره‌گر، شاخص اشاره‌گر آرایه و نماد اشاره‌گر/افست.

i) آدرس مورد مراجعه vPtr + 3 کجاست؟ مقدار ذخیره شده در آن مکان؟

j) فرض کنید که vPtr به values[4] اشاره دارد، آدرس مورد مراجعه توسط vPtr -=4 کجاست؟ مقدار ذخیره شده در آن مکان؟

۸–۱۰ برای هر یک از موارد زیر یک عبارت C++ بنویسید. فرض کنید که متغیرهای صحیح long بنام‌های value1 و value2 قبلا اعلان شده‌اند و value1 با 200000 مقداردهی اولیه شده است.

a) اعلان متغیر longPtr برای اشاره به یک شی از نوع long.

b) تخصیص آدرس متغیر value1 به متغیر اشاره‌گر longPtr.

c) چاپ مقدار شی اشاره شده توسط longPtr.

d) تخصیص مقدار شی مورد اشاره توسط longPtr به متغیر value2.

e) چاپ مقدار value2.

f) چاپ آدرس value1.

g) چاپ آدرس ذخیره شده در longPtr. آیا مقدار چاپ شده همان آدرس value1 است؟

۸–۱۱ برای انجام موارد زیر عباراتی در C++ بنویسید:

a) سرآیند تابع zero را بنویسید که یک مقدار صحیح long از پارامتر آرایه bigIntegers دریافت کرده و مقداری برگشت نمی‌دهد.

b) نمونه اولیه تابع را برای تابع معرفی شده در بخش (a) بنویسید.

c) سرآیند تابع add1AndSum را بنویسید که پارامتر آرایه‌ای صحیح oneTooSmall را دریافت و مقدار صحیح برگشت دهد.

d) نمونه اولیه تابع را برای تابع معرفی شده در بخش (c) بنویسید.

فصل نهم

کلاس‌ها: نگاهی عمیق‌تر: بخش I

اهداف

- نحوه استفاده از یک پوشاننده پیش‌پردازنده برای اجتناب از خطاهای آشکار که با کپی کردن بیش از یکبار فایل سرآیند در فایل کد منبع بوجود می‌آیند.
- آشنایی با مفهوم قلمرو کلاس و دسترسی به اعضاء کلاس از طریق نام یک شی، مراجعه به یک شی یا اشاره‌گر به یک شی.
- تعریف سازنده‌ها با آرگومان‌های پیش‌فرض.
- نحوه استفاده از سازنده‌ها برای انجام عملیات «خاتمه کار» بر روی یک شی قبل از نابود شدن و از بین رفتن آن.
- زمان فراخوانی سازنده‌ها و نابود کننده‌ها و ترتیب فراخوانی آنها.
- خطاهای منطقی که به هنگام برگشت دادن یک مراجعه به داده private توسط یک تابع عضو public رخ می‌دهند.
- انتصاب عضوهای داده یک شی به عضوهای یک شی دیگر با تخصیص Memberwise.

رئوس مطالب

۹-۱ مقدمه

در فصل‌های قبلی، به معرفی برخی از مفاهیم پایه در برنامه‌نویسی شی‌گرا در C++ پرداختیم. همچنین در ارتباط با روش و اسلوب توسعه و ایجاد برنامه‌هایمان صحبت کردیم: صفات و رفتار مقتضی برای هر کلاس را انتخاب می‌کنیم و به یک روش معین مشخص می‌سازیم که کدام شی‌ها از کلاس‌هایمان با شی‌های موجود در کتابخانه کلاس‌های استاندارد C++ برای برآورده کردن هر هدف برنامه می‌توانند همکاری کنند.

در این فصل، نگاهی عمیق‌تر به کلاس‌ها خواهیم داشت. از کلاس یکپارچه **Time** بعنوان یک مبحث آموزشی در این فصل (سه مثال) و فصل دهم (دو مثال) استفاده کرده‌ایم تا به بیان روش‌های ایجاد کلاس بپردازیم. کار را با یک کلاس **Time** شروع می‌کنیم که نگاهی مجدد بر چندین ویژگی عرضه شده در فصل‌های قبلی داشته باشیم. همچنین این مثال به توصیف یک مفهوم اساسی در مهندسی نرم‌افزار C++ یعنی «*پوشاننده پیش پردازنده*» در ارتباط با فایلهای سرآیند می‌پردازد تا از قرار گرفتن بیش از یکبار کد سرآیند در همان فایل کد منبع جلوگیری شود. زمانیکه یک کلاس بتواند فقط یکبار تعریف شود، استفاده از چنین دستوردهنده‌های پیش‌پردازنده از وقوع خطاهای آشکار متعدد جلوگیری می‌کند.

سپس در ارتباط با قلمرو کلاس و رابطه موجود مابین اعضای کلاس صحبت خواهیم کرد. همچنین به توضیح اینکه چگونه کد سرویس‌گیرنده می‌تواند به اعضای **public** کلاس از طریق سه نوع «دستگیره» (نام شی، مراجعه به شی یا اشاره‌گر به شی) دسترسی پیدا کند، خواهیم پرداخت. همانطوری که خواهید

دید، اسامی شی و مراجعه‌ها می‌توانند به همراه عملگر انتخاب عضو (.) در دسترسی به اعضای **public** و اشاره‌گرها می‌توانند با عملگر انتخاب عضو (<-) بکار گرفته شوند. در مورد توابع دسترسی که می‌توانند مبادرت به خواندن یا نمایش داده از یک شی نمایند صحبت خواهیم کرد. یکی از روش‌های رایج در استفاده از توابع دسترسی بررسی شرط‌ها به لحاظ برقرار یا برقرار نبودن (درست و غلط) است، همانند توابعی که بعنوان توابع خبره شناخته می‌شوند. همچنین به بررسی مفهوم و نظریه یک تابع یوتیلیتی (که تابع کمکی هم نامیده می‌شود) می‌پردازیم که یک تابع عضو **private** است که از عملیات توابع عضو کلاس **public** پشتیبانی می‌کند، اما نامزد استفاده توسط سرویس‌گیرنده‌های کلاس نیست.

در دومین مثال از کلاس **Time**، به بررسی نحوه ارسال آرگومان‌ها به سازنده‌ها و نمایش نحوه استفاده از آرگومان پیش‌فرض در یک سازنده می‌پردازیم که به کد سرویس‌گیرنده امکان مقداردهی اولیه شی‌های یک کلاس را با استفاده از آرگومان‌های گوناگون را می‌دهند. سپس در مورد یک تابع عضو خاص بنام سازنده صحبت می‌کنیم که بخشی از هر کلاس بوده و برای انجام «خاتمه کار» بر روی یک شی قبل از اینکه آن شی نابود شود بکار گرفته می‌شود. سپس به بررسی ترتیب فراخوانی سازنده‌ها و نابودکننده‌ها می‌پردازیم، چرا که عملکرد صحیح برنامه بستگی به مقداردهی درست شی‌های دارد که هنوز نابود نشده‌اند. آخرین مثالی که در بحث آموزشی کلاس **Time** در این فصل مطرح شده، به بررسی خطراتی می‌پردازد که از ضعف برنامه‌نویسی حاصل می‌شوند که در این مورد یک تابع عضو یک مراجعه به داده **private** برگشت می‌دهد. همچنین توضیح خواهیم داد که چگونه اینکار می‌تواند سبب از هم‌گسیختگی کپسول کلاس شده و به کد سرویس‌گیرنده اجازه دهد تا بطور مستقیم به داده شی دسترسی پیدا کند. این مثال نشان می‌دهد که شی‌های از یک کلاس می‌توانند با استفاده از تخصیص *memberwise* به دیگری تخصیص داده شوند، که در آن اعضای داده در شی قرار گرفته در سمت راست عملگر تخصیص به اعضای داده متناظر قرار گرفته در سمت چپ عملگر تخصیص، انتساب داده می‌شوند. همچنین این فصل حاوی بحثی در ارتباط با استفاده مجدد از نرم‌افزار است.

۹-۲ مبحث آموزشی: کلاس Time

در اولین مثال (شکل‌های ۹-۳ الی ۹-۱) مبادرت به ایجاد کلاس Time و یک برنامه راه‌انداز برای تست کلاس می‌کنیم. تا بدین مرحله از کتاب چندین کلاس ایجاد کرده‌ایم. در این بخش، نگاهی بر مفاهیم عرضه شده در فصل سوم و بیان اهمیت استفاده از «پوشاننده پیش‌پردازنده» در مهندسی نرم‌افزار ++C خواهیم داشت. زمانیکه کلاسی بتواند فقط یکبار تعریف شود، استفاده از چنین دستوردهنده‌های پیش‌پردازنده از وقوع خطاهای آشکار مضاعف جلوگیری خواهد کرد.

```
// Fig. 9.1: Time.h
// Declaration of class Time.
```

```
3  // Member functions are defined in Time.cpp
4
5  // prevent multiple inclusions of header file
6  #ifndef TIME_H
7  #define TIME_H
8
9  // Time class definition
10 class Time
11 {
12 public:
13    Time(); // constructor
14    void setTime( int, int, int ); // set hour, minute and second
15    void printUniversal(); // print time in universal-time format
16    void printStandard(); // print time in standard-time format
17 private:
18    int hour; // 0 - 23 (24-hour clock format)
19    int minute; // 0 - 59
20    int second; // 0 - 59
21 }; // end class Time
22
23 #endif
```

شکل ۱-۹ | تعریف کلاس Time.

تعریف کلاس Time

تعریف کلاس (شکل ۱-۹) حاوی نمونه اولیه (خطوط 16-13) برای توابع عضو **Time**، **setTime**، **printUniversal** و **printStandard** است. همچنین کلاس شامل اعضای صحیح خصوصی (private) بنام‌های **hour**، **minute** و **second** در خطوط 20-18 می‌باشد. اعضای داده خصوصی **Time** می‌توانند فقط از طریق چهار تابع عضو خود در دسترس قرار گیرند. در فصل ۱۲ به معرفی سومین تصریح کننده دسترسی بنام **protected** خواهیم پرداخت، زمانیکه به بررسی توارث و نقش آن در برنامه‌نویسی شی‌گرا پرداختیم.

برنامه‌نویسی ایده‌ال

برای افزایش خوانایی و وضوح برنامه، از هر تصریح کننده دسترسی فقط یکبار در همه تعریف کلاس استفاده کنید. ابتدا اعضای ***public*** *را قرار دهید، که پیدا کردن آنها آسان باشد.*

مهندسی نرم‌افزار

هر عنصر از کلاس باید در دید private قرار داشته باشد، مگر اینکه مسلم گردد آن عنصر نیاز به میدانی با دید public دارد.

در برنامه شکل ۱-۹ توجه کنید که تعریف کلاس در پوشاننده پیش‌پردازنده احاطه شده است (خطوط 23 ,57):

```
// prevent multiple inclusions of header file
# ifndef TIME_H
# define TIME_H
   …
# endif
```

زمانیکه برنامه‌های بزرگتر ایجاد می‌کنیم، تعاریف و اعلان‌های دیگر هم در فایلهای سرآیند جای داده خواهند شد. پوشاننده پیش‌پردازنده فوق سبب می‌شود تا از تکرار کد مابین **#ifndef** (به معنی "if not

"defined) و **#endif** اجتناب شود اگر شامل نام TIME_H بوده و تعریف شده باشد. اگر سرآیند قبلاً در فایلی بکار گرفته نشده باشد، نام TIME_H توسط دستور دهنده **#define** تعریف شده و فایل سرآیند بکار گرفته خواهد شد. اگر سرآیند قبلاً استفاده شده باشد TIME_H تعریف شده و فایل سرآیند مجدداً در برنامه وارد نخواهد شد.

اجتناب از خطا

*از دستوردهنده‌های پیش‌پردازنده **#ifndef**، **#define** و **#endif** بعنوان پوشاننده پیش‌پردازنده استفاده کنید تا از وارد کردن بیش از یکبار فایلهای سرآیند به برنامه جلوگیری شود.*

برنامه‌نویسی ایده‌ال

*در نام فایل سرآیند از حروف بزرگ به همراه نقطه بجای خط زیرین در دستوردهنده‌های پیش‌پردازنده **#ifndef** و **#define** از یک فایل سرآیند استفاده کنید.*

توابع عضو کلاس Time

در برنامه شکل ۲-۹، سازنده **Time** در خطوط 17-14 مبادرت به مقداردهی اولیه اعضای داده با صفر کرده است (یعنی زمان جهانی معادل با 12 AM). با اینکار مطمئن خواهیم شد که شی کار خود را از یک وضعیت یا حالت پایدار آغاز خواهد کرد. مقادیر اشتباه یا نامعتبر نمی‌توانند در اعضای داده یک شی **Time** ذخیره شوند، چرا که سازنده به هنگام ایجاد شی **Time** فراخوانی شده و تمام فعالیت‌های که توسط یک سرویس‌گیرنده به منظور تغییر دادن اعضای داده صورت می‌گیرد، توسط تابع **setTime** بدقت بررسی می‌شود. توجه به این نکته مهم است که برنامه‌نویس قادر به تعریف چندین سازنده سربارگذاری شده برای یک کلاس باشد.

```
1  // Fig. 9.2: Time.cpp
2  // Member-function definitions for class Time.
3  #include <iostream>
4  using std::cout;
5
6  #include <iomanip>
7  using std::setfill;
8  using std::setw;
9
10 #include "Time.h" // include definition of class Time from Time.h
11
12 // Time constructor initializes each data member to zero.
13 // Ensures all Time objects start in a consistent state.
14 Time::Time()
15 {
16    hour = minute = second = 0;
17 } // end Time constructor
18
19 // set new Time value using universal time; ensure that
20 // the data remains consistent by setting invalid values to zero
21 void Time::setTime( int h, int m, int s )
22 {
23    hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
24    minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
25    second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
26 } // end function setTime
27
```

```
28 // print Time in universal-time format (HH:MM:SS)
29 void Time::printUniversal()
30 {
31    cout << setfill( '0' ) << setw( 2 ) << hour << ":"
32       << setw( 2 ) << minute << ":" << setw( 2 ) << second;
33 } // end function printUniversal
34
35 // print Time in standard-time format (HH:MM:SS AM or PM)
36 void Time::printStandard()
37 {
38    cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ) << ":"
39       << setfill( '0' ) << setw( 2 ) << minute << ":" << setw( 2 )
40       << second << ( hour < 12 ? " AM" : " PM" );
41 } // end function printStandard
```

شکل ۲-۹ | تعریف تابع عضو کلاس Time .

نمی‌توان اعضای داده یک کلاس را در مکانی که در بدنه کلاس اعلان شده‌اند، مقداردهی اولیه کرد. بشدت توصیه می‌شود که این اعضای داده توسط سازنده کلاس مقداردهی اولیه شوند. همچنین می‌توان توسط تابع *set* کلاس **Time** مبادرت به تخصیص مقادیر به داده‌های عضو کرد. [نکته: در فصل دهم نشان خواهیم داد که فقط اعضای داده **static const** یک کلاس از نوع‌های صحیح یا **enum** را می‌توان در بدنه کلاس مقداردهی اولیه کرد.]

خطای برنامه‌نویسی

اقدام به مقداردهی صریح اولیه یک عضو داده غیراستاتیک از یک کلاس در تعریف کلاس، خطای نحوی است.

تابع **setTime** در خطوط 26-21 یک تابع **public** است که سه پارامتر **int** اعلان کرده و از آنها برای تنظیم زمان استفاده می‌کند. یک عبارت شرطی مبادرت به تست هر آرگومان می‌کند تا تعیین کنید که آیا مقدار موجود در محدودهٔ خاص قرار دارد یا خیر. برای مثال، مقدار **hour** در خط 23 بایستی بزرگتر یا برابر صفر و کوچکتر از 24 باشد، چرا که در فرمت جهانی زمان، ساعت یک مقدار صحیح از صفر تا 23 است (برای مثال، 1PM نشاندهنده ساعت 13 و 11PM نشاندهنده 23 است، نیمه شب برابر ساعت 0 و نیمروز برابر ساعت 12 است). به همین ترتیب، مقادیر **minute** و **second** (خطوط 24 و 25) بایستی بزرگتر یا برابر صفر و کمتر از 60 باشند. هر مقداری خارج از این محدودها با صفر تنظیم می‌شود تا مطمئن شویم که شی **Time** همیشه حاوی داده سازگار است، در اینحالت حتی اگر آرگومان‌های ارسالی به تابع **setTime** صحیح نباشند، داده‌های شی همیشه در محدود صحیح نگهداری خواهند شد. در این مثل، صفر یک مقدار سازگار برای **hour**، **minute** و **secand** است. مقدار ارسالی به **setTime** یک مقدار صحیح خواهد بود اگر در محدودهٔ تعیین شده قرار داشته باشد. بنابر این هر عددی در محدوده 23-0 یک مقدار صحیح برای **hour** تلقی خواهد شد. با این همه، یک مقدار سازگار، ضرورتاً نمی‌تواند یک مقدار صحیح باشد. اگر **setTime** مبادرت به تنظیم **hour** با صفر کند به این دلیل که آرگومان دریافتی خارج از

محدودهٔ است، پس فقط **hour** (ساعت) در صورتی صحیح خواهد بود که زمان جاری همزمان با نیمه شب باشد.

تابع **printUniversal** (خطوط 33-29 از شکل ۲-۹) هیچ آرگومانی دریافت نمی‌کند و تاریخ را برحسب فرمت جهانی زمان، متشکل از سه جفت کولن متمایز کننده ارقام برای ساعت، دقیقه و ثانیه چاپ می‌کند. برای مثال اگر زمان 1:30:07PM باشد، تابع **printUniversal** مقدار 13:30:07 برگشت می‌دهد. دقت کنید که در خط 31 از تابع **setfill** برای مشخص کردن کاراکتر پرکننده استفاده شده است که مبادرت به نمایش یک مقدار صحیح در خروجی در یک فیلد بجای *ارزش عددی* از ارقام می‌کند. بطور پیش‌فرض، کاراکترهای پرکننده در سمت چپ ارقام یک عدد ظاهر می‌شوند. در این مثال، اگر مقدار **minute** (دقیقه) برابر 2 باشد، بصورت 02 به نمایش در خواهد آمد، چرا که کاراکتر پرکننده با صفر (**'0'**) تنظیم شده است. اگر عددی که قرار است در خروجی قرار گیرد کل فیلد تعیین شده را در برگیرد، کاراکتر پرکننده به نمایش در نخواهد آمد. توجه کنید زمانیکه کاراکتر پرکننده با **setfill** همراه می‌شود، این کاراکتر بر روی مابقی مقادیری که در پهنای آن فیلد به نمایش در خواهند آمد، هم اعمال خواهد گردید. نقطه مقابل آن **setw** است که فقط بر روی مقدار بعدی به نمایش در آمده اعمال می‌شود.

تابع **printStandard** (خطوط 41-36) هیچ آرگومانی دریافت نمی‌کند و تاریخ را در فرمت استاندارد زمانی به نمایش در می‌آورد. این فرمت متشکل از مقادیر ساعت، دقیقه و ثانیه است که توسط کولن‌های که بدنبال آن AM یا PM قرار دارد از هم جدا شده‌اند (مانند 1:27:06 PM). همانند تابع **printUniversal** تابع **printStandard** از **setfill('0')** برای قالب‌بندی دقیقه و ثانیه بعنوان دو مقدار رقمی با دنباله صفر در صورت نیاز استفاده کرده است. در خط 38 از عملگر شرطی (**:?**) برای تعیین نمایش مقدار ساعت استفاده شده است. اگر ساعت برابر 0 یا 12 (AM یا PM) باشد بصورت 12 و در غیر اینصورت ساعت بصورت مقداری از 1 تا 11 به نمایش در خواهد آمد. عملگر شرطی بکار رفته در خط 40 تعیین می‌کند که آیا AM یا PM به نمایش درآید یا خیر.

تعریف توابع عضو خارج از تعریف کلاس: قلمرو کلاس

حتی در صورتیکه یک تابع عضو در تعریف کلاسی اعلان شده باشد می‌تواند در خارج از تعریف کلاس، تعریف گردد (و به کلاس از طریق عملگر باینری تفکیک قلمرو مرتبط شود)، که هنوز هم تابع عضو در درون قلمرو کلاس قرار خواهد داشت، به این معنی که نام تابع فقط توسط اعضای دیگر کلاس شناخته شده خواهد بود مگر اینکه از طریق شیی از کلاس مورد مراجعه قرار گیرد یا اشاره‌گری به یک شی از کلاس یا عملگر باینری تفکیک قلمرو بکار گرفته شود.

اگر تابع عضوی در بدنه یک کلاس تعریف شده باشد، کامپایلر C++ مبادرت به فراخوانی inline آن تابع خواهد کرد. توابع عضو تعریف شده در خارج از تعریف کلاس می‌توانند بصورت صریح و خطی و توسط کلمه کلیدی ***inline*** فراخوانی شوند.

توابع عضو در مقابل توابع سراسری

نکته جالب توجه در این است که توابع عضو **printUniversal** و **printStandard** هیچ آرگومانی دریافت نمی‌کنند. به این دلیل که این توابع عضو بصورت غیرصریح می‌دانند که باید اعضای داده شی **Time** را به هنگام فعال شدن چاپ کنند. چنین عملی فراخوانی توابع عضو را به نسبت توابع عادی در برنامه نویسی روالی بسیار مختصر می‌کند.

استفاده از کلاس Time

پس از اینکه کلاس **Time** تعریف شد، می‌توان از آن بعنوان یک نوع در اعلان شی، آرایه و اشاره‌گر بصورت زیر استفاده کرد:

```
Time sunset; // object of type Time
Time arrayOfTimes [5], // array of 5 Time objects
Time &dinnerTime = swnset; // reference to a Time object
Time *timePtr = &dinnerTime, // pointer to a Time object
```

در برنامه شکل ۳-۹ از کلاس **Time** استفاده شده است. خط 12 مبادرت به نمونه‌سازی یک شی منفرد از کلاس **Time** بنام **t** می‌کند. زمانیکه یک شی معرفی می‌شود، سازنده **Time** برای مقداردهی اولیه هر داده عضو **private** با صفر فراخوانی می‌شود سپس، خطوط 16 و 18 زمان را با فرمت‌های استاندارد و جهانی برای تایید اینکه اعضا بدرستی مقداردهی اولیه شده‌اند چاپ می‌کنند. خط 20 مبادرت به تنظیم زمان جدید با فراخوانی تابع عضو **setTime** کرده و خطوط 24 و 26 مجدداً زمان را در هر دو فرمت چاپ می‌کنند. خط 28 مبادرت به استفاده از تابع **setTime** برای تنظیم اعضای داده با مقادیر اشتباه می‌کند. در اینحالت تابع **setTime** این اشتباه را تشخیص داده و مقادیر اشتباه را با صفر تنظیم می‌نماید تا شی در یک وضعیت سازگار (پایدار) باقی بماند. سرانجام، خطوط 33 و 35 زمان را در هر دو فرمت چاپ می‌کنند.

نگاهی جلوتر ترکیب و توارث

غالباً، کلاس‌ها مجبور نیستند از «ابتدا» ایجاد شوند. بجای آن می‌توانند حاوی شی‌های از کلاس‌های دیگر بعنوان اعضا باشند یا می‌تواند از کلاس‌های دیگر مشتق شوند تا صفات و رفتارهای برای استفاده کلاس‌های جدید فراهم آورند. چنین قابلیتی که بعنوان استفاده مجدد از نرم‌افزار شناخته می‌شود می‌تواند در نگهداری کد و کارایی برنامه‌نویسی نقش مهمی بازی کند. وارد کردن شی‌های کلاس بعنوان اعضای کلاس‌های دیگر، *ترکیب* (یا *تجمع*) نامیده می‌شود و در فصل دهم توضیح داده شده است. مشتق کردن کلاس‌های جدید از کلاس‌های موجود، *توارث* نامیده می‌شود و در فصل دوازدهم به توضیح آن پرداخته شده است.

```
1  // Fig. 9.3: fig09_03.cpp
2  // Program to test class Time.
3  // NOTE: This file must be compiled with Time.cpp.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  #include "Time.h" // include definition of class Time from Time.h
9
10 int main()
11 {
12    Time t; // instantiate object t of class Time
13
14    // output Time object t's initial values
15    cout << "The initial universal time is ";
16    t.printUniversal(); // 00:00:00
17    cout << "\nThe initial standard time is ";
18    t.printStandard(); // 12:00:00 AM
19
20    t.setTime( 13, 27, 6 ); // change time
21
22    // output Time object t's new values
23    cout << "\n\nUniversal time after setTime is ";
24    t.printUniversal(); // 13:27:06
25    cout << "\nStandard time after setTime is ";
26    t.printStandard(); // 1:27:06 PM
27
28    t.setTime( 99, 99, 99 ); // attempt invalid settings
29
30    // output t's values after specifying invalid values
31    cout << "\n\nAfter attempting invalid settings:"
32       << "\nUniversal time: ";
33    t.printUniversal(); // 00:00:00
34    cout << "\nStandard time: ";
35    t.printStandard(); // 12:00:00 AM
36    cout << endl;
37    return 0;
38 } // end main
```

```
The initial universal time is 00:00:00
The initial standard time is 12:00:00 AM

Universal time after setTime is 13:27:06
Standard time after setTime is 1:27:06 PM

After attempting invalid settings:
Universal time: 00:00:00
Standard time: 12:00:00 AM
```

شکل ۳-۹ | برنامه تست کلاس Time.

سایزشی

افراد کم تجربه در برنامه‌نویسی شی‌گرا تصور می‌کنند که بایستی شی‌ها بکلی بزرگ باشند چرا که آنها حاوی اعضای داده و توابع عضو می‌باشند. به لحاظ منطقی این تصور صحیح است، و برنامه‌نویس می‌تواند چنین ذهنیتی داشته باشد. با این همه، به لحاظ فیزیکی این امر صادق نیست.

۳-۹ قلمرو کلاس و دسترسی به اعضاء کلاس

اعضای داده کلاس (متغیرهای اعلان شده در تعریف کلاس) و توابع عضو (توابع اعلان شده در تعریف کلاس) به قلمرو کلاس تعلق دارند. توابع غیرعضو در *قلمرو فایل* جای دارند.

در درون قلمرو یک کلاس، اعضای کلاس بلادرنگ توسط تمام توابع عضو کلاس در دسترس بوده و می‌تواند توسط نام مورد مراجعه قرار گیرند. خارج از قلمرو کلاس، اعضای کلاس **public** از طریق یک دستگیره یا هندل به شی، مورد مراجعه قرار می‌گیرند. نوع شی، مراجعه یا اشاره‌گر تصریح کننده واسط دسترس پذیر برای سرویس‌گیرنده هستند.

توابع عضو یک کلاس می‌توانند سربارگذاری شوند، اما فقط توسط توابع عضو دیگر آن کلاس چنین کاری امکان‌پذیر است. برای سربارگذاری یک تابع عضو، کافیست در تعریف کلاس یک نمونه اولیه برای هر نسخه از تابع سربارگذاری شده تدارک دید و یک تعریف تابع مجزا برای هر نسخه از تابع در نظر گرفت.

متغیرهای اعلان شده در یک تابع عضو دارای *قلمرو بلوکی* بوده و فقط در آن تابع شناخته می‌شوند. اگر یک تابع عضو، متغیری با همان نام بعنوان متغیر با قلمرو کلاس تعریف نماید، متغیر قرار گرفته در قلمرو کلاس توسط متغیر قلمرو بلوکی در قلمرو بلوک پنهان خواهد شد. به چنین متغیر پنهان شده‌ای می‌توان با قرار دادن نام متغیر قبل از نام کلاس به همراه عملگر تفکیک قلمرو (:) دسترسی پیدا کرد. متغیرهای پنهان شده سراسری می‌توانند با استفاده از عملگر غیرباینری تفکیک قلمرو در دسترس قرار گیرند (فصل ششم).

با استفاده از عملگر انتخاب عضو (.) قبل از نام یک شی یا با مراجعه به یک شی می‌توان به اعضای شی دسترسی پیدا کرد. استفاده از عملگر انتخاب عضو (->) قبل از یک اشاره‌گر به یک شی می‌توان به اعضای شی دسترسی پیدا کرد.

در برنامه شکل ۴-۹ از یک کلاس ساده بنام **Count** در خطوط 25-8 به همراه عضو داده **private** بنام **x** از نوع **int** (خط 24)، تابع عضو **public** بنام **setX** (خطوط 15-12) و تابع عضو **public** بنام **print** (خطوط 21-18) استفاده شده تا به توضیح نحوه دسترسی به اعضای یک کلاس با استفاده از عملگرهای انتخاب عضو بپردازیم. برای ساده‌تر شدن موضوع، این کلاس کوچک را در همان فایل تابع **main** قرار داده‌ایم که از آن استفاده کند. در خطوط 31-29 سه متغیر مرتبط با نوع **Count** بنام‌های **counter** (یک شی **Count**)، **counterPtr** (یک اشاره‌گر به شی **Count**) و **counterRef** (یک مراجعه به شی **Count**) ایجاد شده است. متغیر **counterRef** به **counter** مراجعه دارد و متغیر **counterPtr** به **counter** اشاره می‌کند. در خطوط 35-34 و 39-38 توجه کنید که برنامه می‌تواند توابع عضو **setX** و **print** را با استفاده از عملگر نقطه انتخاب عضو (.) به همراه نام شی (**counter**) یا مراجعه‌ای به شی (**counterRef** که نام دیگر **counter** است) فراخوانی کند. به همین ترتیب، خطوط 43-42 نشان می‌دهند که برنامه می‌تواند توابع عضو **setX** و **print** را با استفاده از یک اشاره‌گر (**countPtr**) و عملگر انتخاب عضو (->) فراخوانی نماید.

```
1  // Fig. 9.4: fig09_04.cpp
2  // Demonstrating the class member access operators . and ->
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  // class Count definition
8  class Count
9  {
10 public: // public data is dangerous
11    // sets the value of private data member x
12    void setX( int value )
13    {
14       x = value;
15    } // end function setX
16
17    // prints the value of private data member x
18    void print()
19    {
20       cout << x << endl;
21    } // end function print
22
23 private:
24    int x;
25 }; // end class Count
26
27 int main()
28 {
29    Count counter; // create counter object
30    Count *counterPtr = &counter; // create pointer to counter
31    Count &counterRef = counter; // create reference to counter
32
33    cout << "Set x to 1 and print using the object's name: ";
34    counter.setX( 1 ); // set data member x to 1
35    counter.print(); // call member function print
36
37    cout << "Set x to 2 and print using a reference to an object: ";
38    counterRef.setX( 2 ); // set data member x to 2
39    counterRef.print(); // call member function print
40
41    cout << "Set x to 3 and print using a pointer to an object: ";
42    counterPtr->setX( 3 ); // set data member x to 3
43    counterPtr->print(); // call member function print
44    return 0;
45 } // end main
```

```
Set x to 1 and print using the object's name: 1
Set x to 2 and print usinga reference to an object: 2
Set x to 3 and print using a pointer to an object: 3
```

شکل ٤-٩ | دسترسی به توابع عضو یک شی از طریق نوع شی.

٤-٩ جداسازی واسط از پیاده‌سازی

در فصل سوم، شروع به وارد کردن تعریف کلاس و تعریف تابع عضو در یک فایل کردیم. سپس به توضیح نحوه جداسازی این کد به دو فایل پرداختیم، یک فایل سرآیند برای تعریف کلاس (یعنی واسط کلاس) و فایل کد منبع برای تعریف تابع عضو کلاس (یعنی پیاده‌سازی کلاس). بخاطر دارید که با انجام اینکار انجام تغییرات در برنامه‌ها آسانتر می‌شود، تا آنجا که به سرویس‌گیرندگان کلاس مربوط می‌شود، تغییر در پیاده‌سازی کلاس تاثیری در سرویس‌گیرنده ندارد تا مادامیکه واسط تدارک دیده شده توسط کلاس برای سرویس‌گیرنده بدون تغییر باقی مانده باشد.

البته اینکار به همین سادگی هم نیست. سرآیند حاوی برخی از قسمت‌های پیاده‌سازی بوده و اشاره بسیار جزئی به دیگران دارند. برای مثال، توابع عضو **Inline** (خطی)، نیاز دارند در یک فایل سرآیند قرار داشته باشند، از اینروست که به هنگام کامپایل شدن یک سرویس‌گیرنده، سرویس‌گیرنده می‌تواند حاوی تعریف تابع *inline* باشد. اعضای **private** یک کلاس در فایل سرآیند تعریف کلاس لیست می‌شوند، از اینروست که این اعضا در دید سرویس‌گیرنده‌ها قرار دارند حتی اگر سرویس‌گیرنده‌ها قادر به دسترسی به اعضای **private** نباشند. در فصل دهم، با نحوه استفاده از «کلاس پروکسی» برای پنهان کردن داده **private** یک کلاس از دید سرویس‌گیرنده‌ها آشنا خواهید شد.

۵-۹ توابع دسترسی و توابع یوتیلیتی

توابع دسترسی قادر به خواندن و نمایش داده‌ها هستند. یکی دیگر از کاربردهای رایج توابع دسترسی در تست برقراری یا عدم برقراری شرط‌ها است، به چنین توابعی، *توابع پیشگو* یا *مسند* می‌گویند. مثالی از یک تابع مسند می‌تواند تابع **isEmpty** برای هر کلاس حامل باشد، کلاسی که قادر به نگهداری شی‌های متعدد است، نظیر یک لیست پیوندی، یک پشته یا صف. برنامه می‌تواند با تست **isEmpty** قبل از مبادرت به خواندن ایتم دیگری از شی حامل، اطمینان حاصل کند. می‌توان از تابع مسند **isFull** برای تست یک کلاس حامل استفاده کرده و تعیین کرد که آیا دارای فضای اضافی هست یا خیر. توابع مسند مناسب برای کلاس **Time** می‌تواند **isAM** و **isPM** باشد.

برنامه بکار رفته در شکل‌های ۷-۹ الی ۵-۹ به توضیح مفهوم یک تابع یوتیلیتی (*تابع کمکی* هم نامیده می‌شود) می‌پردازد. یک تابع یوتیلیتی بخشی از واسط **public** یک کلاس نیست، ترجیحاً یک تابع عضو **private** است که از عملیات توابع عضو کلاس **public** پشتیبانی می‌کند. توابع یوتیلیتی نامزد استفاده از سرویس‌گیرنده‌های یک کلاس نیستند (اما می‌توانند توسط **friend** یک کلاس بکار گرفته شوند، همانطوری‌که در فصل دهم شاهد خواهید بود). کلاس **SalesPerson** (شکل ۵-۹) یک آرایه 12 عنصری از فروش دوازده ماهه (خط 16) و نوع اولیه برای سازنده کلاس و توابع عضو که آرایه را دستکاری می‌کنند، اعلان کرده است.

```
1  // Fig. 9.5: SalesPerson.h
2  // SalesPerson class definition.
3  // Member functions defined in SalesPerson.cpp.
4  #ifndef SALESP_H
5  #define SALESP_H
6
7  class SalesPerson
8  {
9  public:
10    SalesPerson(); // constructor
11    void getSalesFromUser(); // input sales from keyboard
12    void setSales( int, double ); // set sales for a specific month
13    void printAnnualSales(); // summarize and print sales
14 private:
15    double totalAnnualSales(); // prototype for utility function
16    double sales[ 12 ]; // 12 monthly sales figures
```

```
17 }; // end class SalesPerson
18
19 #endif
```

شکل ۵-۹ | تعریف کلاس SalesPerson.

در برنامه شکل ۶-۹ سازنده **SalesPerson** (خطوط 19-15) مبادرت به مقداردهی اولیه آرایه **sales** با صفر کرده است. تابع عضو سراسری **setSales** (خطوط 43-36) مبادرت به تنظیم فروش برای یک ماه در آرایه **sales** می‌کند. تابع عضو سراسری **public** بنام **printAnnualSales** (خطوط 51-46) مجموع فروش دوازده ماهه را چاپ می‌کند. تابع یوتیلیتی خصوصی (private) بنام **totalAnnualSales** (خطوط 62-54) مجموع فروش دوازده ماهه را با استفاده از **printAnnualSales** بدست می‌آورد. در برنامه شکل ۷-۹، توجه کنید که کاربرد تابع **main** فقط در فراخوانی پشت سرهم توابع عضو بوده و دارای عبارات کنترلی نمی‌باشد. منطق کار با آرایه **sales** در این است که این آرایه بطور کامل در توابع عضو کلاس **SalesPerson** کپسوله شود.

```
1  // Fig. 9.6: SalesPerson.cpp
2  // Member functions for class SalesPerson.
3  #include <iostream>
4  using std::cout;
5  using std::cin;
6  using std::endl;
7  using std::fixed;
8
9  #include <iomanip>
10 using std::setprecision;
11
12 #include "SalesPerson.h" // include SalesPerson class definition
13
14 // initialize elements of array sales to 0.0
15 SalesPerson::SalesPerson()
16 {
17    for ( int i = 0; i < 12; i++ )
18       sales[ i ] = 0.0;
19 } // end SalesPerson constructor
20
21 // get 12 sales figures from the user at the keyboard
22 void SalesPerson::getSalesFromUser()
23 {
24    double salesFigure;
25
26    for ( int i = 1; i <= 12; i++ )
27    {
28       cout << "Enter sales amount for month " << i << ": ";
29       cin >> salesFigure;
30       setSales( i, salesFigure );
31    } // end for
32 } // end function getSalesFromUser
33
34 // set one of the 12 monthly sales figures; function subtracts
35 // one from month value for proper subscript in sales array
36 void SalesPerson::setSales( int month, double amount )
37 {
38    // test for valid month and amount values
39    if ( month >= 1 && month <= 12 && amount > 0 )
40       sales[ month - 1 ] = amount; // adjust for subscripts 0-11
41    else // invalid month or amount value
42       cout << "Invalid month or sales figure" << endl;
43 } // end function setSales
44
45 // print total annual sales (with the help of utility function)
```

```
46 void SalesPerson::printAnnualSales()
47 {
48    cout << setprecision( 2 ) << fixed
49       << "\nThe total annual sales are: $"
50       << totalAnnualSales() << endl; // call utility function
51 } // end function printAnnualSales
52
53 // private utility function to total annual sales
54 double SalesPerson::totalAnnualSales()
55 {
56    double total = 0.0; // initialize total
57
58    for ( int i = 0; i < 12; i++ ) // summarize sales results
59       total += sales[ i ]; // add month i sales to total
60
61    return total;
62 } // end function totalAnnualSales
```

شکل ۶-۹ | تعریف تابع عضو کلاس SalesPerson .

```
1  // Fig. 9.7: fig09_07.cpp
2  // Demonstrating a utility function.
3  // Compile this program with SalesPerson.cpp
4
5  // include SalesPerson class definition from SalesPerson.h
6  #include "SalesPerson.h"
7
8  int main()
9  {
10    SalesPerson s; // create SalesPerson object s
11
12    s.getSalesFromUser(); // note simple sequential code;
13    s.printAnnualSales(); // no control statements in main
14    return 0;
15 } // end main
```

```
Enter sales amount for month 1: 5314.76
Enter sales amount for month 2: 4292.38
Enter sales amount for month 3: 4589.83
Enter sales amount for month 4: 5534.03
Enter sales amount for month 5: 4376.34
Enter sales amount for month 6: 5698.45
Enter sales amount for month 7: 4439.22
Enter sales amount for month 8: 5893.57
Enter sales amount for month 9: 4909.67
Enter sales amount for month 10: 5123.45
Enter sales amount for month 11: 4024.97
Enter sales amount for month 12: 5923.92

The total annual sales are: $60120.59
```

شکل ۷-۹ | تابع یوتیلیتی.

۶-۹ مبحث آموزشی کلاس Time : سازنده‌ها همراه با آرگومان‌های پیش‌فرض

برنامه بکار رفته در شکل‌های ۸-۹ الی ۱۰-۹ کارایی کلاس **Time** را با توضیح نحوه ارسال آرگومان بصورت غیرصریح به یک سازنده افزایش داده‌اند. سازنده تعریف شده در شکل ۲-۹ مبادرت به مقداردهی اولیه ساعت، دقیقه و ثانیه با صفر می‌کند (یعنی نیمه شب در فرمت جهانی). همانند توابع دیگر، سازنده‌ها می‌توانند تعیین کننده آرگومان‌های پیش‌فرض باشند. خط 13 از برنامه شکل ۸-۹ سازنده **Time** را که شامل آرگومان‌های پیش‌فرض است اعلان کرده است که مشخص کننده یک مقدار پیش‌فرض صفر برای هر آرگومان ارسالی به سازنده است. در شکل ۹-۹، خطوط 17-14 یک نسخه جدید از سازنده

Time تعریف کرده‌اند که مقادیری برای پارامترهای **hr**، **min** و **sec** دریافت می‌کند که در مقداردهی اولیه اعضاء داده ساعت، دقیقه و ثانیه کاربرد دارند.

```
1  // Fig. 9.8: Time.h
2  // Declaration of class Time.
3  // Member functions defined in Time.cpp.
4
5  // prevent multiple inclusions of header file
6  #ifndef TIME_H
7  #define TIME_H
8
9  // Time abstract data type definition
10 class Time
11 {
12 public:
13    Time( int = 0, int = 0, int = 0 ); // default constructor
14
15    // set functions
16    void setTime( int, int, int ); // set hour, minute, second
17    void setHour( int ); // set hour (after validation)
18    void setMinute( int ); // set minute (after validation)
19    void setSecond( int ); // set second (after validation)
20
21    // get functions
22    int getHour(); // return hour
23    int getMinute(); // return minute
24    int getSecond(); // return second
25
26    void printUniversal(); // output time in universal-time format
27    void printStandard(); // output time in standard-time format
28 private:
29    int hour; // 0 - 23 (24-hour clock format)
30    int minute; // 0 - 59
31    int second; // 0 - 59
32 }; // end class Time
33
34 #endif
```

شکل ۸-۹ | کلاس Time حاوی یک سازنده با آرگومان‌های پیش‌فرض.

توجه کنید که کلاس **Time** مبادرت به تدارک دیدن توابع *set* و *get* برای هر عضو داده کرده است. اکنون سازنده **Time** اقدام به فراخوانی **setTime** می‌کند و آن هم توابع **setHour**، **setMinute** و **setSecond** را برای اعتبار سنجی و تخصیص مقادیر به اعضای داده فراخوانی می‌کند. آرگومان‌های پیش‌فرض، سازنده را مطمئن می‌سازند که حتی اگر مقادیر در فراخوانی سازنده در نظر گرفته نشده باشند، سازنده قادر به مقداردهی اولیه اعضای داده باشد تا بتواند شی **Time** را در یک وضعیت پایدار نگهداری کند. سازنده‌ای که تمام آرگومان‌های آن پیش‌فرض هستند، یک سازنده پیش‌فرض محسوب می‌شود، یعنی سازنده‌ای که می‌تواند بدون آرگومان فراخوانی یا فعال گردد. حداکثر یک سازنده پیش‌فرض در هر کلاس می‌تواند وجود داشته باشد.

```
1  // Fig. 9.9: Time.cpp
2  // Member-function definitions for class Time.
3  #include <iostream>
4  using std::cout;
5
6  #include <iomanip>
7  using std::setfill;
8  using std::setw;
9
```

```
10 #include "Time.h" // include definition of class Time from Time.h
11
12 // Time constructor initializes each data member to zero;
13 // ensures that Time objects start in a consistent state
14 Time::Time( int hr, int min, int sec )
15 {
16    setTime( hr, min, sec ); // validate and set time
17 } // end Time constructor
18
19 // set new Time value using universal time; ensure that
20 // the data remains consistent by setting invalid values to zero
21 void Time::setTime( int h, int m, int s )
22 {
23    setHour( h ); // set private field hour
24    setMinute( m ); // set private field minute
25    setSecond( s ); // set private field second
26 } // end function setTime
27
28 // set hour value
29 void Time::setHour( int h )
30 {
31    hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
32 } // end function setHour
33
34 // set minute value
35 void Time::setMinute( int m )
36 {
37    minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
38 } // end function setMinute
39
40 // set second value
41 void Time::setSecond( int s )
42 {
43    second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
44 } // end function setSecond
45
46 // return hour value
47 int Time::getHour()
48 {
49    return hour;
50 } // end function getHour
51
52 // return minute value
53 int Time::getMinute()
54 {
55    return minute;
56 } // end function getMinute
57
58 // return second value
59 int Time::getSecond()
60 {
61    return second;
62 } // end function getSecond
63
64 // print Time in universal-time format (HH:MM:SS)
65 void Time::printUniversal()
66 {
67    cout << setfill( '0' ) << setw( 2 ) << getHour() << ":"
68       << setw( 2 ) << getMinute() << ":" << setw( 2 ) << getSecond();
69 } // end function printUniversal
70
71 // print Time in standard-time format (HH:MM:SS AM or PM)
72 void Time::printStandard()
73 {
74    cout << (( getHour() == 0 || getHour() == 12) ? 12: getHour() % 12 )
75       << ":" << setfill( '0' ) << setw( 2 ) << getMinute()
76       << ":" << setw(2)<< getSecond() << ( hour < 12 ? " AM" : " PM" );
77 } // end function printStandard
```

شکل ۹-۹ | تعریف تابع عضو کلاس Time شامل یک سازنده که آرگومان دریافت می‌کند.

در خط 16 از شکل ۹-۹، سازنده مبادرت به فراخوانی از تابع عضو **setTime** با مقادیر ارسالی به سازنده می‌کند (یا مقادیر پیش‌فرض). تابع **setTime** تابع **setHour** را فراخوانی می‌کند تا ممطئن گردد که مقدار تدارک دیده شده برای ساعت در بازه 0-23 قرار دارد، سپس تابع **setMinute** و **setSecond** برای اطمینان از اینکه مقادیر تدارک دیده شده برای دقیقه و ثانیه نیز در بازه 0-59 جای دارند، فراخوانی می‌شوند. اگر مقداری خارج از محدوده باشد، آن مقدار با صفر تنظیم می‌شود.

دقت کنید که سازنده **Time** می‌توانست برای در برگرفتن همان عبارات بعنوان تابع عضو **setTime** یا حتی عبارات جداگانه در توابع **setHour**، **setMinute** و **setSecond** نوشته شود. فراخوانی **setHour**، **setMinute** و **setSecond** از طریق سازنده می‌تواند کمی موثرتر واقع شود چرا که می‌تواند فراخوانی زیاد **setTime** را برطرف سازد و حذف نماید. به همین ترتیب، کپی کد از خطوط 37، 31 و 43 بدون سازنده می‌تواند هزینه فراخوانی **setTime**، **setHour**، **setMinute** و **setSecond** را کاهش دهد. کد نویسی سازنده **Time** یا تابع عضو **setTime** بعنوان کپی از کد در خطوط 37، 31 و 43 می‌تواند نگهداری این کلاس را بسیار سخت نماید. اگر پیاده‌سازی **setHour**، **setMinute** و **setSecond** دچار تغییر شود، پیاده‌سازی هر تابع عضو که در خطوط 37، 31 و 43 تکرار شده‌اند هم متعاقب آن تغییر خواهند یافت.

با مجبور کردن سازنده **Time** برای فراخوانی **setTime** و مجبور کردن **setTime** برای فراخوانی **setHour**، **setMinute** و **setSecond** امکان می‌دهد تا نیاز به تغییر کدی که مبادرت به اعتبار سنجی ساعت، دقیقه و ثانیه می‌کند، به حداقل برسد. همچنین کارایی سازنده **Time** و **setTime** می‌تواند با اعلان صریح آنها بصورت *inline* یا تعریف آنها در تعریف کلاس افزایش یابد.

مهندسی نرم‌افزار: هر تغییری در مقادیر آرگومان پیش‌فرض یک تابع مستلزم کامپایل مجدد کد سرویس گیرنده است.

مهندسی نرم‌افزار

هر تغییری در مقادیر آرگومان پیش‌فرض یک تابع مستلزم کامپایل مجدد کد سرویس گیرنده است.

تابع **main** در شکل ۱۰-۹ اقدام به مقداردهی اولیه پنج شی **Time** می‌کند. یکی با سه آرگومان پیش‌فرض در فراخوانی غیرصریح سازنده (خط 11)، یکی با یک آرگومان مشخص شده (خط 12)، یکی با دو آرگومان مشخص شده (خط 13)، یکی با سه آرگومان مشخص شده (خط 14) و یکی با سه آرگومان اشتباه مشخص شده در خط 15. سپس برنامه هر شی را در فرمت زمانی استاندارد و جهانی به نمایش در می‌آورد.

```
1  // Fig. 9.10: fig09_10.cpp
2  // Demonstrating a default constructor for class Time.
3  #include <iostream>
```

```
4  using std::cout;
5  using std::endl;
6
7  #include "Time.h" // include definition of class Time from Time.h
8
9  int main()
10 {
11    Time t1; // all arguments defaulted
12    Time t2( 2 ); // hour specified; minute and second defaulted
13    Time t3( 21, 34 ); // hour and minute specified; second defaulted
14    Time t4( 12, 25, 42 ); // hour, minute and second specified
15    Time t5( 27, 74, 99 ); // all bad values specified
16
17    cout << "Constructed with:\n\nt1: all arguments defaulted\n  ";
18    t1.printUniversal(); // 00:00:00
19    cout << "\n  ";
20    t1.printStandard(); // 12:00:00 AM
21
22    cout << "\n\nt2: hour specified; minute and second defaulted\n  ";
23    t2.printUniversal(); // 02:00:00
24    cout << "\n  ";
25    t2.printStandard(); // 2:00:00 AM
26
27    cout << "\n\nt3: hour and minute specified; second defaulted\n  ";
28    t3.printUniversal(); // 21:34:00
29    cout << "\n  ";
30    t3.printStandard(); // 9:34:00 PM
31
32    cout << "\n\nt4: hour, minute and second specified\n  ";
33    t4.printUniversal(); // 12:25:42
34    cout << "\n  ";
35    t4.printStandard(); // 12:25:42 PM
36
37    cout << "\n\nt5: all invalid values specified\n  ";
38    t5.printUniversal(); // 00:00:00
39    cout << "\n  ";
40    t5.printStandard(); // 12:00:00 AM
41    cout << endl;
42    return 0;
43 } // end main
```

```
Constructed with:

t1: all arguments defaulted
   00:00:00
   12:00:00 AM

t2: hour specified; minute and second defaulted
   02:00:00
   2:00:00 AM

t3: hour and minute specified; second defaulted
  12:34:00
  9:34:00 PM

t4: hour, minute and second defaulted
  12:25:42
  12:25:42 AM

t5: all invalid values specified
  00:00:00
  12:00:00 AM
```

شکل ۱۰-۹ | سازنده با آرگومان‌های پیش‌فرض.

۹-۷ نابود کننده‌ها

نابودکننده (*تخریب‌کننده*) نوع دیگری از تابع عضو می‌باشد. نام نابودکننده یک کلاس همراه با کاراکتر مد (~) و نام کلاس مشخص می‌شود. این قاعده نامگذاری دارای جاذبه شهودی است، زیرا همانطوری که

در یک فصل پایانی خواهید دید، عملگر مد یک عملگر *مکمل بیتی* است و تا اندازه‌ای یک نابودکننده، متمم یک سازنده است. توجه کنید که غالبا از نابودکننده‌ها در مقالات بعنوان "dtor" یاد می‌شود. ما ترجیح می‌دهیم که از این کلمه استفاده نکنیم. نابودکننده یک کلاس بصورت تلویحی (غیرصریح) و در زمان از بین رفتن شی فراخوانی می‌شود. برای مثال، این اتفاق برای شیی رخ می‌دهد که برنامه از قلمرو که شی در آن ایجاد شده خارج گردد. توجه کنید که خود نابودکننده نمی‌تواند حافظه اخذ شده توسط شی را آزاد سازد، این تابع عملیات خاتمه کار، قبل از اینکه سیستم حافظه شی را بازپس بگیرد وارد صحنه می‌شود.

نابودکننده پارامتر دریافت نمی‌کند و مقداری برگشت نمی‌دهد. نابودکننده نوع خاصی را هم برگشت نمی‌دهد (حتی void). یک کلاس می‌تواند فقط یک نابودکننده داشته باشد. نابودکننده را نمی‌توان سربارگذاری کرد.

خطای برنامه‌نویسی

ارسال آرگومان به یک نابودکننده، تعیین نوع برگشتی به یک نابودکننده، برگشت دادن مقدار از یک نابودکننده یا سربارگذاری آن خطای نحوی است.

با اینکه تا بدین جا برای کلاس‌های معرفی شده، نابودکننده تدارک ندیده‌ایم، اما هر کلاسی دارای یک نابودکننده است. اگر برنامه‌نویس بطور صریح اقدام به تدارک دیدن یک نابودکننده نکند، خود کامپایلر یک نابودکننده تهی ایجاد می‌کند. در فصل یازدهم، اقدام به ایجاد نابودکننده‌های متناسب با کلاس‌هایی خواهیم که شی‌های آنها حاوی حافظه اخذ شده دینامیکی هستند (همانند آرایه‌ها و رشته‌ها) یا از منابع دیگر سیستم استفاده می‌کنند (همانند فایلها که در فصل هفدهم به بررسی آنها خواهیم پرداخت).

۸-۹ زمان فراخوانی سازنده‌ها و نابودکننده‌ها

سازنده‌ها و نابودکننده‌ها بصورت غیرصریح توسط کامپایلر فراخوانی می‌شوند. ترتیب فراخوانی این توابع بستگی به‌ترتیب ورود آنها به مرحله اجرا و ترک قلمرو دارد که شی‌ها در آن نمونه‌سازی شده‌اند. بطور کلی، فراخوانی نابودکننده‌ها به ترتیب معکوس از فراخوانی سازنده‌های مقتضی صورت می‌گیرد، اما همانطوری که در برنامه‌های شکل ۱۱-۹ الی۱۳-۹ شاهد خواهید بود، کلاس‌های ذخیره‌سازی شی‌ها می‌توانند ترتیب فراخوانی نابودکننده‌ها را در دچار تغییر سازند.

فراخوانی سازنده‌های متعلق به شی‌های تعریف شده در قلمرو سراسری قبل از هر تابعی (شامل main هم می‌شود) صورت می‌گیرند. نابودکننده‌های متناظر پس از اتمام **main** فراخوانی می‌شوند. تابع **exit** برنامه را مجبور می‌کند تا بلافاصله و بدون اجرای نابودکننده بر روی شی‌های اتوماتیک خاتمه یابد. از این تابع اغلب برای خاتمه دادن به برنامه در زمان‌های که خطایی در ورودی مشاهده شود یا اینکه فایل مورد نظر

برای پردازش باز نشود استفاده می‌شود. تابع **abort** کار مشابهی با تابع **exit** انجام می‌دهد، اما برنامه را بلافاصله مجبور به خاتمه می‌کند بدون اینکه به نابودکننده‌ای هر شی اجازه دهد تا فراخوانی گردند. معمولاً از تابع **abort** برای تشخیص خاتمه غیرعادی برنامه استفاده می‌شود.

سازنده برای یک شی محلی اتوماتیک زمانی فراخوانی می‌شود که اجرا برنامه به مکانی برسد که شی در آنجا تعریف شده است، نابودکننده متناظر هم زمانی فراخوانی می‌گردد که اجرا، قلمرو شی را ترک می‌کند. سازنده‌ها و نابودکننده‌های متعلق به شی‌های اتوماتیک در هر بار ورود و خروج اجرای برنامه به قلمرو شی فراخوانی می‌شوند. نابودکننده‌ها برای شی‌های اتوماتیک فراخوانی نمی‌شوند اگر برنامه با فراخوانی تابع **exit** یا **abort** خاتمه یابد.

سازنده برای یک شی محلی **static** (استاتیک) فقط یک بار فراخوانی می‌شود و آن هم زمانی است که اجرا به مکانی برسد که شی در آنجا تعریف شده است. نابودکننده متناظر هم زمانی فراخوانی می‌شود که **main** خاتمه یافته باشد یا برنامه، تابع **exit** را فراخوانی کرده باشد. شی‌های سراسری و استاتیک به ترتیب معکوس از ایجاد خود نابود می‌شوند. نابودکننده‌ها در صورتیکه برنامه با فراخوانی تابع **abort** خاتمه یافته باشد، برای شی‌های استاتیک فراخوانی نخواهند شد. برنامه بکار رفته در شکل‌های۱۳-۹ الی۱۱-۹ نشاندهنده ترتیبی است که سازنده‌ها و نابودکننده‌های برای شی‌های کلاس **CreateAndDestory** فراخوانی می‌شوند (شکل ۱۱-۹ و شکل ۱۲-۹) از کلاس‌های ذخیره‌سازی مختلف در قلمروهای گوناگون. هر شی از کلاس **CreateAndDestory** حاوی (خطوط 17-16) یک نوع صحیح (objectID) و یک رشته (message) است که در خروجی برنامه بکار گرفته شده‌اند تا هویت شی باشند. این مثال صرفاً جنبه آموزشی دارد. از اینرو، خط 23 از نابودکننده در شکل ۱۲-۹ تعیین می‌کند که آیا شی نابود شده دارای شناسه (objectID) با مقدار 1 یا 6 است یا خیر، و اگر چنین باشد یک کاراکتر خط جدید در خروجی قرار داده می‌شود. این خط کمک می‌کند تا درک خروجی برنامه آسانتر شود.

```
1  // Fig. 9.11: CreateAndDestroy.h
2  // Definition of class CreateAndDestroy.
3  // Member functions defined in CreateAndDestroy.cpp.
4  #include <string>
5  using std::string;
6
7  #ifndef CREATE_H
8  #define CREATE_H
9
10 class CreateAndDestroy
11 {
12 public:
13    CreateAndDestroy( int, string ); // constructor
14    ~CreateAndDestroy(); // destructor
15 private:
16    int objectID; // ID number for object
17    string message; // message describing object
18 }; // end class CreateAndDestroy
19
20 #endif
```

شکل ۱۱-۹ | تعریف کلاس CreateAndDestory.

```
1  // Fig. 9.12: CreateAndDestroy.cpp
2  // Member-function definitions for class CreateAndDestroy.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "CreateAndDestroy.h" // include CreateAndDestroy class definition
8
9  // constructor
10 CreateAndDestroy::CreateAndDestroy( int ID, string messageString )
11 {
12    objectID = ID; // set object's ID number
13    message = messageString; // set object's descriptive message
14
15    cout << "Object " << objectID << "   constructor runs   "
16       << message << endl;
17 } // end CreateAndDestroy constructor
18
19 // destructor
20 CreateAndDestroy::~CreateAndDestroy()
21 {
22    // output newline for certain objects; helps readability
23    cout << ( objectID == 1 || objectID == 6 ? "\n" : "" );
24
25    cout << "Object " << objectID << "   destructor runs    "
26       << message << endl;
27 } // end ~CreateAndDestroy destructor
```

شکل ۱۲-۹ | تعریف تابع عضو کلاس CreateAndDestory.

در شکل ۱۳-۹ شی **first** در قلمرو سراسری تعریف شده است (خط 12). در واقع سازنده آن قبل از اجرای هر عبارتی در **main** فراخوانی می‌شود و نابودکننده آن در خاتمه برنامه و پس از اینکه نابودکننده سایر شی‌ها اجرا شدند، فراخوانی می‌گردد.

تابع **main** (خطوط 26-14) سه شی اعلان کرده است. شی‌های **second** (خط 17) و **fourth** (خط 23) شی‌های اتوماتیک محلی بوده و شی **third** (خط 18) یک شی محلی استاتیک است. سازنده هر یک از این شی‌ها به هنگام رسیدن اجرا به نقطه‌ای که شی در آن اعلان شده فراخوانی می‌شود. نابودکننده شی‌های **fourth** و سپس **second** زمانی فراخوانی می‌شوند که اجرا به انتهای **main** رسیده باشد. بدلیل اینکه شی **third** استاتیک است، تا زمان خاتمه برنامه باقی می‌ماند. نابودکننده شی **third** قبل از نابودکننده شی سراسری **first**، اما پس از اینکه تمام شی‌های دیگر نابود شدند فراخوانی می‌گردد.

تابع **create** (خطوط 36-29) سه شی اعلان کرده است، **fifth** (خط 32) و **seventh** (خط 34) بعنوان شی‌های اتوماتیک محلی و **sixth** (خط 33) بعنوان یک شی استاتیک محلی. نابودکننده شی‌های **seventh** و سپس **fifth** فراخوانی می‌شوند زمانیکه **create** خاتمه می‌پذیرد. بدلیل اینکه **sixth** استاتیک است تا زمان خاتمه برنامه باقی می‌ماند. نابودکننده **sixth** قبل از نابودکننده **third** و **first** فراخوانی می‌شود، اما پس از نابودی تمام شی‌های دیگر نابود می‌شود.

```
1  // Fig. 9.13: fig09_13.cpp
2  // Demonstrating the order in which constructors and
3  // destructors are called.
4  #include <iostream>
```

```
5  using std::cout;
6  using std::endl;
7
8  #include "CreateAndDestroy.h" // include CreateAndDestroy class definition
9
10 void create( void ); // prototype
11
12 CreateAndDestroy first( 1, "(global before main)" ); // global object
13
14 int main()
15 {
16    cout << "\nMAIN FUNCTION: EXECUTION BEGINS" << endl;
17    CreateAndDestroy second( 2, "(local automatic in main)" );
18    static CreateAndDestroy third( 3, "(local static in main)" );
19
20    create(); // call function to create objects
21
22    cout << "\nMAIN FUNCTION: EXECUTION RESUMES" << endl;
23    CreateAndDestroy fourth( 4, "(local automatic in main)" );
24    cout << "\nMAIN FUNCTION: EXECUTION ENDS" << endl;
25    return 0;
26 } // end main
27
28 // function to create objects
29 void create( void )
30 {
31    cout << "\nCREATE FUNCTION: EXECUTION BEGINS" << endl;
32    CreateAndDestroy fifth( 5, "(local automatic in create)" );
33    static CreateAndDestroy sixth( 6, "(local static in create)" );
34    CreateAndDestroy seventh( 7, "(local automatic in create)" );
35    cout << "\nCREATE FUNCTION: EXECUTION ENDS" << endl;
36 } // end function create
```

```
Object 1   constructor     runs     (global before main)

MAIN FUNCTION: EXECUTION BEGINS
Object 2       constructor  runs    (local automatic in main)
Object 3       constructor  runs    (local static in main)

CREATE FUNCTION: EXECUTION BEGINS
Object 5       constructor  runs    (local automatic in create)
Object 6       constructor  runs    (local static in create)
Object 7       constructor  runs    (local automatic in create)

CREATE FUNCTION: EXECUTION ENDS
Object 7       destructor   runs    (local automatic in create)
Object 5       destructor   runs    (local automatic in create)

MAIN FUNCTION: EXECUTION RESUMES
Object 4       constructor  runs    (local automatic in main)

MAIN FUNCTION: EXECUTION ENDS
Object 4       destructor   runs    (local automatic in main)
Object 2       destructor   runs    (local automatic in main)

Object 6       destructor   runs    (local static in create)
Object 3       destructor   runs    (local static in main)

Object 1       destructor   runs    (global before main)
```

شکل ۱۳-۹ | ترتیب فراخوانی سازنده‌ها و نابودکننده‌ها.

۹-۹ مبحث آموزشی کلاس Time : برگشت دادن یک مراجعه به داده عضو private

یک مراجعه به یک شی نام مستعار برای نام شی بوده و از اینرو، می‌تواند در سمت چپ یک عبارت تخصیص بکار گرفته شود. در این زمینه، مراجعه‌ها بخوبی پذیرای نقش lvalue هستند و می‌توانند مقداری را دریافت کنند. یک روش استفاده از این قابلیت (متاسفانه) داشتن یک تابع عضو **public** از کلاسی است

که یک مراجعه به یک عضو داده **private** از آن کلاس برگشت می‌دهد. دقت کنید که اگر تابعی یک مراجعه ثابت (**const**) برگشت دهد، از آن مراجعه نمی‌توان به‌عنوان یک lvalue اصلاح‌پذیر استفاده کرد. در برنامه شکل‌های ۱۶-۹ الی ۱۴-۹ از یک کلاس ساده شده **Time** (شکل ۱۴-۹ و ۱۵-۹) استفاده شده تا به بررسی برگشت دادن یک مراجعه به یک داده عضو **private** با تابع عضو **badSetHour** (اعلان شده در شکل ۱۴-۹ از خط 15 و تعریف شده در شکل ۱۵-۹ از خطوط 33-29) پرداخته شود. در واقع برگشت دادن چنین مراجعه‌ای سبب فراخوانی تابع عضو **badSetHour** بعنوان نام جانشین برای عضو داده خصوصی **hour** می‌کند. می‌توان از فراخوانی تابع به هر روشی استفاده کرد که در آن عضو داده **private** (خصوصی) می‌تواند حتی بعنوان یک lvalue در یک عبارت تخصیص بکار گرفته شود، از اینرو سرویس‌گیرنده‌های کلاس قادر خواهند بود تا داده **private** کلاس را بطور دلخواه پاک کنند (دستکاری). توجه کنید که همین مشکل می‌تواند در صورتیکه یک اشاره‌گر به داده **private** توسط تابعی برگشت داده شود، رخ دهد.

```
1  // Fig. 9.14: Time.h
2  // Declaration of class Time.
3  // Member functions defined in Time.cpp
4
5  // prevent multiple inclusions of header file
6  #ifndef TIME_H
7  #define TIME_H
8
9  class Time
10 {
11 public:
12    Time( int = 0, int = 0, int = 0 );
13    void setTime( int, int, int );
14    int getHour();
15    int &badSetHour( int ); // DANGEROUS reference return
16 private:
17    int hour;
18    int minute;
19    int second;
20 }; // end class Time
21
22 #endif
```

شکل ۱۴-۹ | برگشت دادن یک مراجعه به داده عضو private.

```
1  // Fig. 9.15: Time.cpp
2  // Member-function definitions for Time class.
3  #include "Time.h" // include definition of class Time
4
5  // constructor function to initialize private data;
6  // calls member function setTime to set variables;
7  // default values are 0 (see class definition)
8  Time::Time( int hr, int min, int sec )
9  {
10    setTime( hr, min, sec );
11 } // end Time constructor
12
13 // set values of hour, minute and second
14 void Time::setTime( int h, int m, int s )
15 {
16    hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
17    minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
18    second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
```

```
19 } // end function setTime
20
21 // return hour value
22 int Time::getHour()
23 {
24    return hour;
25 } // end function getHour
26
27 // POOR PROGRAMMING PRACTICE:
28 // Returning a reference to a private data member.
29 int &Time::badSetHour( int hh )
30 {
31    hour = ( hh >= 0 && hh < 24 ) ? hh : 0;
32    return hour; // DANGEROUS reference return
33 } // end function badSetHour
```

شکل ۱۵-۹ | برگشت دادن یک مراجعه به یک داده عضو private.

در برنامه شکل ۱۶-۹ یک شی **Time** بنام **t** (خط 12) و یک مراجعه بنام **hourRef** (خط 15) اعلان شده است که با مراجعه برگشتی توسط فراخوانی **t.badSetHour(20)** مقداردهی اولیه می‌شود. خط 17 مقدار مستعار **hourRef** را نشان می‌دهد. با این عمل نشان داده می‌شود که چگونه **hourRef** ویژگی کپسوله‌سازی کلاس را شکسته است، عبارات موجود در **main** نبایستی به داده **private** کلاس دسترسی داشته باشند. سپس، خط 18 از نام مستعار برای تنظیم مقدار **hour** با 30 استفاده کرده (یک مقدار نامعتبر) و خط 19 مقدار برگشتی توسط تابع **getHour** را برای نمایش اینکه مقدار تخصیص‌ی به **hourRef** واقعاً داده **private** در شی **t** را تغییر داده است، به نمایش در می‌آورد. سرانجام، خط 23 از فراخوانی خود تابع **badSetHour** بعنوان یک lvalue استفاده کرده و 74 (یک مقدار نامعتبر دیگر) به مراجعه برگشتی توسط تابع تخصیص می‌دهد.

```
1  // Fig. 9.16: fig09_16.cpp
2  // Demonstrating a public member function that
3  // returns a reference to a private data member.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  #include "Time.h" // include definition of class Time
9
10 int main()
11 {
12    Time t; // create Time object
13
14    // initialize hourRef with the reference returned by badSetHour
15    int &hourRef = t.badSetHour( 20 ); // 20 is a valid hour
16
17    cout << "Valid hour before modification: " << hourRef;
18    hourRef = 30; // use hourRef to set invalid value in Time object t
19    cout << "\nInvalid hour after modification: " << t.getHour();
20
21    // Dangerous: Function call that returns
22    // a reference can be used as an lvalue!
23    t.badSetHour( 12 ) = 74; // assign another invalid value to hour
24
25    cout << "\n\n*********************************************\n"
26       << "POOR PROGRAMMING PRACTICE!!!!!!!!\n"
27       << "t.badSetHour( 12 ) as an lvalue, invalid hour: "
28       << t.getHour()
29       << "\n*********************************************" << endl;
30    return 0;
31 } // end main
```

```
Valid hour before modification: 20
Invalid hour after modification: 30

*************************************************
POOR PROGRAMMING PRACTICE!!!!!!!!
t.badSetHour( 12 ) as an lvalue, invalid hour: 74
*************************************************
```

شکل ۹-۱۶ | برگشت دادن یک مراجعه به یک عضو داده private.

خط 28 مجدداً مقدار برگشتی توسط تابع **getHour** را برای نمایش اینکه مقدار تخصیص یافته به نتیجه فراخوانی تابع در خط 23 داده **private** در شی **t** را تغییر داده است، به نمایش در می‌آورد.

۹-۱۰ تخصیص Memberwise

می‌توان از عملگر تخصیص (=) برای تخصیص یک شی به شی دیگر از همان نوع استفاده کرد. بطور پیش‌فرض، چنین تخصیصی توسط *تخصیص memberwise* صورت می‌گیرد، هر عضو داده از شی در سمت راست عملگر تخصیص بطور جداگانه به همان عضو داده در شی قرار گرفته در سمت چپ عملگر تخصیص، انتساب داده می‌شود. در شکل‌های ۹-۱۷ و ۹-۱۸ کلاس **Date** برای استفاده در این مثال تعریف شده است. در خط 20 از شکل ۹-۱۹ از تخصیص memberwie برای انتساب اعضای داده **date1** از کلاس **Date** به اعضای داده متناظر **date2** از کلاس **Date** استفاده شده است.

```
1  // Fig. 9.17: Date.h
2  // Declaration of class Date.
3  // Member functions are defined in Date.cpp
4
5  // prevent multiple inclusions of header file
6  #ifndef DATE_H
7  #define DATE_H
8
9  // class Date definition
10 class Date
11 {
12 public:
13    Date( int = 1, int = 1, int = 2000 ); // default constructor
14    void print();
15 private:
16    int month;
17    int day;
18    int year;
19 }; // end class Date
20
21 #endif
```

شکل ۹-۱۷ | فایل سرآیند کلاس Date.

```
1  // Fig. 9.18: Date.cpp
2  // Member-function definitions for class Date.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Date.h" // include definition of class Date from Date.h
8
9  // Date constructor (should do range checking)
10 Date::Date( int m, int d, int y )
11 {
12    month = m;
13    day = d;
14    year = y;
```

```
15 } // end constructor Date
16
17 // print Date in the format mm/dd/yyyy
18 void Date::print()
19 {
20    cout << month << '/' << day << '/' << year;
21 } // end function print
```

شکل ۱۸-۹ | تعریف عضو داده کلاس Date.

```
1  // Fig. 9.19: fig09_19.cpp
2  // Demonstrating that class objects can be assigned
3  // to each other using default memberwise assignment.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  #include "Date.h" // include definition of class Date from Date.h
9
10 int main()
11 {
12    Date date1( 7, 4, 2004 );
13    Date date2; // date2 defaults to 1/1/2000
14
15    cout << "date1 = ";
16    date1.print();
17    cout << "\ndate2 = ";
18    date2.print();
19
20    date2 = date1; // default memberwise assignment
21
22    cout << "\n\nAfter default memberwise assignment, date2 = ";
23    date2.print();
24    cout << endl;
25    return 0;
26 } // end main
```

```
date1 = 7/4/2004
date2 = 1/1/2000

After default memberwise assignment, date2 = 7/4/2004
```

شکل ۱۹-۹ | تخصیص memberwise.

در این مورد، عضو **month** از **date1** به عضو **month** از **date2**، عضو **day** از **date1** به عضو **day** از **date2** و عضو **year** از **date1** به عضو **year** از **date2** تخصیص می‌یابد. توجه کنید که سازنده **Date** حاوی هیچ بخشی برای بررسی خطا نیست.

شی‌ها می‌توانند بعنوان آرگومان‌های تابع ارسال شده و می‌توانند از توابع برگشت داده شوند. چنین ارسال و برگشتی بطور پیش‌فرض به *روش ارسال با مقدار* صورت می‌گیرد که در آن یک کپی از شی ارسال یا برگشت داده می‌شود. در چنین حالتی، C++ یک شی جدید ایجاد و از یک سازنده کپی کننده برای کپی مقدار شی اصلی به شی جدید استفاده می‌کند. برای هر کلاسی، کامپایلر یک سازنده کپی‌کننده پیش‌فرض تدارک دیده که هر عضو از شی اصلی را به عضو متناظر در شی جدید کپی می‌کند. همانند تخصیص memberwise، سازنده‌های کپی کننده می‌توانند به هنگام استفاده با کلاسی که حاوی اشاره‌گرهای با حافظه اخذ شده دینامیکی هستند، مشکل‌ساز شوند. در فصل یازدهم به بررسی این موضوع خواهیم پرداخت.

۹-۱۱ استفاده مجدد از نرم‌افزار

سعی افرادی که سرگرم نوشتن برنامه‌های شی‌گرا هستند، پیاده‌سازی کلاس‌های سودمند و کاربردی‌تر است. انگیزه بسیار چشمگیری وجود دارد که از کلاس‌های تدارک دیده شده توسط جامعه‌های برنامه‌نویسی استفاده شود. تعدادی زیادی از کتابخانه‌های کلاس وجود دارند و برخی در سرتاسر جهان در حال ایجاد می‌باشند. بایستی نرم‌افزار از همان آغاز کار، خوش تعریف، بدقت تست شده، بخوبی مستند شده، قابل حمل با کارایی بالا و از کامپونت‌های قابل دسترس ایجاد شده باشد. چنین نرم‌افزاری با قابلیت استفاده مجدد سرعت نرم‌افزارهای قدرتمند و با کیفیت بالا را افزایش می‌دهد. توسعه سریع برنامه‌های کاربردی (RAD) بواسطه مکانیزم قابل استفاده بودن مجدد اجزاء اهمیت خاصی پیدا کرده است.

مسائل علمی باید حل شوند، اما قبل از آن باید به بررسی دقیق مسائل استفاده مجدد از نرم‌افزار پرداخت. نیاز به فهرست کردن طرح، اخذ مجوز طرح‌ها، مکانیزم‌های حفاظتی برای اطمینان از اینکه کپی‌های اصلی از کلاس‌ها معیوب و خراب تهیه نخواهد شد. توصیف طرح‌ها را داریم تا طراحان سیستم‌های جدید بتواند به آسانی تعیین کنند که آیا شی‌های موجود می‌توانند نیاز آنها را برآورده سازند. همچنین نیاز به مکانیزم مرور داریم تا کلاس‌های موجود و در دسترس را مشخص کرده و نشان دهد که کدام کلاس به خواست طراح نرم‌افزار نزدیکتر است.

۹-۱۲ مبحث آموزشی مهندسی نرم‌افزار: شروع برنامه‌نویسی کلاس‌های سیستم ATM

در بخش‌های مبحث آموزشی مهندسی نرم‌افزار در فصل‌های یک الی هفتم، به معرفی اصول و مفاهیم بنیادین شی‌گرا و طراحی شی‌گرا بر روی سیستم ATM پرداخته شد. در ابتدای این فصل هم به بررسی برخی از جزئیات برنامه‌نویسی کلاس‌ها در ++C پرداختیم. حال شروع به پیاده‌سازی طرح شی‌گرای خود در ++C می‌کنیم. در انتهای این بخش، شما را با نحوه تبدیل دیاگرام‌های کلاس به فایل‌های سرآیند ++C آشنا خواهیم کرد. در بخش پایانی «مبحث آموزشی مهندسی نرم‌افزار» (بخش ۱۰-۱۳)، این فایل‌های سرآیند را برای هماهنگی با مفهوم ارث‌بری در برنامه‌نویسی شی‌گرا اصلاح خواهیم کرد.

رویت

می‌خواهیم تصریح‌کننده‌های دسترسی به اعضای کلاس‌ها را فراهم آوریم. در فصل سوم، به معرفی تصریح‌کننده‌های دسترسی **public** و **private** پرداختیم. این تصریح‌کننده‌ها قابل رویت یا دسترس پذیر بودن صفات و عملیات یک شی را که در اختیار شی‌های دیگر هستند، تعیین می‌کنند. قبل از اینکه بتوانیم شروع به پیاده‌سازی طرح خود نمائیم، بایستی بررسی کنیم که کدام صفات و عملیاتی از کلاس‌ها حالت **public** دارند و کدامیک حالت **private**. در فصل سوم، مشاهده کردید که معمولاً اعضای داده بایستی **private** باشند و آن دسته از توابع عضو که توسط سرویس‌گیرنده‌ها فعال می‌شوند بایستی از نوع **public**

تعیین شوند. توابع عضو که فقط توسط سایر توابع عضو یک کلاس فراخوانی می‌گردند، بعنوان «توابع یوتیلیتی» شناخته می‌شوند، با این همه، معمولاً باید **private** باشند. زبان UML از *نشانگر رویت* برای مدل کردن میزان رویت صفات و عملیات استفاده می‌کند. رویت عمومی (public) با قرار دادن یک نماد جمع (+) قبل از یک عملیات یا صفت شناخته می‌شود. رویت خصوصی (private) هم با یک نماد منفی (-) تعیین می‌گردد. در شکل ۲۰-۹ دیاگرام کلاس با اعمال نشانگرهای رویت به روز شده است. [نکته: در شکل ۲۰-۹ تمام پارامترهای عملیاتی لحاظ نشده است و این کاملاً عادی است. افزودن نشانگرهای رویت تاثیری در پارامترهای مدل شده در دیاگرام‌های کلاس در شکل‌های ۲۲-۶ الی ۲۵-۶ ندارد.]

هدایت

قبل از اینکه شروع به پیاده‌سازی طرح خود با C++ کنیم، به معرفی یکی دیگر از نمادهای UML می‌پردازیم. دیاگرام کلاس در شکل ۲۱-۹ با در اختیار گرفتن *فلش‌های هدایت* به همراه خطوط ارتباطی در میان کلاس‌های سیستم ATM به روز شده است. فلش‌های هدایت نشان می‌دهند که کدام جهت ارتباطی را می‌توان طی کرد که بر پایه مدل همکاری و دیاگرام‌های توالی است (بخش ۱۲-۷). به هنگام پیاده‌سازی یک سیستم طراحی شده با استفاده از UML، برنامه‌نویسان از فلش‌های هدایت برای کمک در تعیین اینکه کدام شی‌ها نیاز به مراجعه یا اشاره به سایر شی‌ها دارند، استفاده می‌کنند. برای مثال فلش هدایت که از کلاس **ATM** به کلاس **BankDatabase** اشاره می‌کند بر این نکته دلالت دارد که می‌توانیم از ابتدا به انتها حرکت کنیم، و در نتیجه **ATM** قادر به فعال کردن عملیات **BankDatabase** می‌شود. با این وجود، در حالیکه شکل ۲۱-۹ حاوی یک فلش هدایت از کلاس **BankDatabase** به کلاس **ATM** نیست، پس **BankDatabase** قادر به دسترسی به عملیات **ATM** نمی‌باشد. دقت کنید که وابستگی‌های موجود در یک دیاگرام کلاس که دارای فلش‌های هدایت در هر دو انتهای خود هستند یا دارای این فلش‌های نیستند، نشاندهنده هدایت دو طرفه (دو سویه) می‌باشند.

شکل ۲۰-۹ | دیاگرام کلاس با نشانگرهای رویت.

همانند دیاگرام کلاس در شکل ۲۳-۳، دیاگرام کلاس در شکل ۲۱-۹ برای حفظ سادگی کلاس‌های **BalanceInquiry** و **Deposit** را در نظر نگرفته است. هدایت اعمال شده در این کلاس‌ها بسیار به هدایت اعمال شده در کلاس **Withdrawal** نزدیک است. از بخش ۱۱-۳ بخاطر دارید که **BalanceInquiry** دارای یک رابطه با کلاس **Screen** است. می‌توانیم از طریق این رابطه از کلاس **BalanceInquiry** به کلاس **Screen** برسیم، اما نمی‌توانیم از کلاس **Screen** به کلاس **BalanceInquiry** هدایت شویم. از اینرو، اگر به سراغ مدل کردن کلاس **BalanceInquiry** در شکل ۲۱-۹ برویم، می‌توانیم یک فلش هدایت در انتهای کلاس **Screen** این رابطه قرار دهیم. همچنین بخاطر دارید یک کلاس **Deposit** با

کلاس‌های **Screen**، **Keypad** و **DepositSlot** رابطه دارد. می‌توانیم از کلاس **Deposit** به هر کدامیک از این کلاس‌ها هدایت شویم، اما عکس اینحالت صادق نیست. از اینرو فلش‌های هدایت را در انتهای رابطه با این کلاس‌ها قرار داده‌ایم.

شکل ۲۱-۹ | دیاگرام کلاس با فلش‌های هدایت.

پیاده‌سازی سیستم ATM از روی طرح UML آن

اکنون آماده هستیم تا شروع به پیاده‌سازی سیستم ATM نمائیم. ابتدا کلاس‌های موجود در دیاگرام‌های شکل‌های ۲۰-۹ و ۲۱-۹ را به فایلهای سرآیند ++C تبدیل می‌کنیم. این کد عرضه کننده «اسکلت» سیستم خواهد بود. در فصل سیزدهم، فایلهای سرآیند را برای بهره‌گیری از مفهوم ارث‌بری اصلاح خواهیم کرد.

بعنوان یک مثال، شروع به ایجاد فایل سرآیند متعلق به کلاس **Withdrawal** از روی طرح موجود این کلاس در شکل ۲۰-۹ می‌کنیم. از این تصویر برای تعیین صفات و عملیات کلاس استفاده کنیم. از مدل UML در شکل ۲۱-۹ برای تعیین وابستگی‌های موجود مابین کلاس‌ها استفاده می‌کنیم. برای هر کلاسی پنج مرحله زیر را دنبال می‌کنیم:

۱ـ از نام قرار گرفته در بخش اول یک کلاس در دیاگرام کلاس برای تعریف کلاس در فایل سرآیند استفاده می‌کنیم (شکل ۲۲-۹). از دستوردهنده‌های پیش‌پردازندهٔ **#ifndef**، **#define** و **#endif** برای اجتناب از اعمال بیش از یکبار فایل سرآیند در برنامه استفاده کنید.

۲ـ از صفات موجود در بخش دوم کلاس برای اعلان اعضای داده استفاده کنید. برای مثال، صفات **private** از کلاس **Withdrawal** عبارتند از **accountNumber** و **amount** که حاصل آن در شکل ۲۳-۹ آورده شده است.

۳- از وابستگی توصیف شده در دیاگرام کلاس برای اعلان مراجعه‌ها (یا اشاره‌گرها در صورت نیاز) به شی‌های دیگر استفاده کنید. برای مثال، مطابق شکل ۲۱-۹، کلاس **Withdrawal** می‌تواند به یک شی از کلاس **Screen**، یک شی از کلاس **Keypad**، یک شی از کلاس **CashDispenser** و یک شی از کلاس **BankDatabase** دسترسی داشته باشد. کلاس **Withdrawal** باید مبادرت به حفظ هندل‌هایی (دستگیره) به این شی‌ها کند، تا پیغام‌های به آنها ارسال نماید. از اینرو خطوط 22-19 از شکل ۲۴-۹ مبادرت به اعلان چهار مراجعه بعنوان اعضای داده **private** کرده‌اند. در پیاده‌سازی **Withdrawal** در ضمیمه G، یک سازنده این اعضای داده را با مراجعه‌های به شی‌های واقعی مقداردهی اولیه کرده است. توجه کنید که در خطوط 9-6، **#include** فایلهای سرآیند حاوی تعاریفی از کلاس‌های **Screen**، **Keypad**، **CashDispenser** و **BankDatabase** است، از اینروست که می‌توانیم مراجعه‌های به شی‌هایی از این کلاس‌ها در خطوط 22-19 اعلان کنیم.

۴- ازدحام ایجاد شده از وارد کردن فایلهای سرآیند کلاس‌های **Screen**، **Keypad**، **CashDispenser** و **BankDatabase** در شکل ۲۴-۹ بیش از نیاز است. کلاس **Withdrawal** حاوی مراجعه‌های به شی‌های از این کلاس‌ها است (حاوی شی‌های واقعی نیست) و مقدار اطلاعات مورد نیاز توسط کامپایلر برای ایجاد یک مراجعه با ایجاد یک شی تفاوت دارد. بخاطر دارید که ایجاد یک شی مستلزم آن است که برای کامپایلر تعریفی از کلاسی که نام کلاس را بعنوان یک نوع جدید تعریف شده از سوی کاربر معرفی می‌کند و نشاندهنده اعضای داده‌ای است که تعیین کننده میزان حافظه مورد نیاز برای آن شی هستند، تدارک دیده باشید. با این همه، اعلان یک مراجعه (یا اشاره‌گر) به یک شی، فقط مستلزم آن است که کامپایلر بداند که شیی از کلاس موجود است و نیازی به دانستن سایز شی ندارد. هر مراجعه (یا اشاره‌گری) صرفنظر از اینکه به کدام شی از کلاسی مراجعه دارد، فقط حاوی آدرس حافظه شی واقعی است. میزان حافظه مورد نیاز برای ذخیره‌سازی یک آدرس یک مسئله سخت افزاری مرتبط با کامپیوتر است. کامپایلر از سایر هر مراجعه یا اشاره‌گری مطلع است. در نتیجه، به هنگام اعلان فقط یک مراجعه به یک شی از آن کلاس، وارد کردن کل فایل سرآیند کلاس ضرورتی ندارد و نیاز به معرفی نام کلاس داریم اما نیازی به تدارک دیدن آرایش داده شی نداریم چرا که کامپایلر از سایز تمام مراجعه‌ها اطلاع دارد. زبان C++ دارای دستوری بنام *اعلان رو به جلو* یا *forward* است که نشان می‌دهد یک فایل سرآیند حاوی مراجعه‌ها یا اشاره‌گرهای به یک کلاس است، اما تعریف کلاس خارج از فایل سرآیند قرار دارد. می‌توانیم **#include**های موجود در تعریف کلاس **Withdrawal** شکل ۲۴-۹ را با اعلان‌های رو به جلو کلاس‌های **Screen**، **Keypad**، **CashDispenser** و **BankDatabase** جایگزین کنیم (خطوط 9-6 در شکل ۲۵-۹). بجای وارد کردن کل فایل سرآیند برای هر کدامیک از این کلاس‌ها، فقها یک اعلان رو به جلو از هر کلاس در فایل سرآیند را برای کلاس **Withdrawal** جایگزین می‌کنیم. توجه کنید که اگر کلاس **Withdrawal** حاوی شی‌های واقعی بجای مراجعه‌ها باشد (یعنی علامت‌های **&** در خطوط 22-19 حذف شوند)، نیاز خواهد بود تا کل فایل‌های سرآیند را وارد (#include) نمائیم.

```
1  // Fig. 9.22: Withdrawal.h
2  // Definition of class Withdrawal that represents a withdrawal transaction.
3  #ifndef WITHDRAWAL_H
4  #define WITHDRAWAL_H
5
6  class Withdrawal
7  {
8  }; // end class Withdrawal
9
10 #endif // WITHDRAWAL_H
```

شکل ۲۲-۹ | تعریف کلاس Withdrawal احاطه شده در پوشاننده‌های پیش‌پردازنده.

```
1  // Fig. 9.23: Withdrawal.h
2  // Definition of class Withdrawal that represents a withdrawal transaction.
3  #ifndef WITHDRAWAL_H
4  #define WITHDRAWAL_H
5
6  class Withdrawal
```

```
7  {
8  private:
9     // attributes
10    int accountNumber; // account to withdraw funds from
11    double amount; // amount to withdraw
12
13 }; // end class Withdrawal
14
15 #endif // WITHDRAWAL_H
```

شکل ۲۳-۹ | افزودن صفات به فایل سرآیند کلاس Withdrawal.

```
1  // Fig. 9.24: Withdrawal.h
2  // Definition of class Withdrawal that represents a withdrawal transaction.
3  #ifndef WITHDRAWAL_H
4  #define WITHDRAWAL_H
5
6  #include "Screen.h" // Screen class definition
7  #include "BankDatabase.h" // BankDatabase class definition
8  #include "Keypad.h" // Keypad class definition
9  #include "CashDispenser.h" // CashDispenser class definition
10
11 class Withdrawal
12 {
13 private:
14    // attributes
15    int accountNumber; // account to withdraw funds from
16    double amount; // amount to withdraw
17
18    // references to associated objects
19    Screen &screen; // reference to ATM's screen
20    Keypad &keypad; // reference to ATM's keypad
21    CashDispenser &cashDispenser; // reference to ATM's cash dispenser
22    BankDatabase &bankDatabase; // reference to the account info database
23 }; // end class Withdrawal
24
25 #endif // WITHDRAWAL_H
```

شکل ۲۴-۹ | اعلان مراجعه‌ها به شی‌های مرتبط با کلاس Withdrawal.

توجه کنید که استفاده از اعلان رو به جلو (تا حد امکان) بجای وارد کردن کل فایل سرآیند از یک مشکل پیش‌پردازنده بنام *وارد گردن دایره‌ای* (*circular include*) جلوگیری می‌کند. این مشکل زمانی برای فایل سرآیند کلاس A رخ می‌دهد که فایل سرآیندی برای کلاس B را include# کرده باشد و برعکس برخی از پیش‌پردازنده‌ها قادر به رفع چنین دستوردهنده‌ها یا رهنمودهای include# نیستند و در نتیجه یک خطای کامپایل رخ می‌دهد. برای مثال، اگر کلاس A فقط از یک مراجعه به یک شی از کلاس B استفاده نماید، پس include# در فایل سرآیند کلاس A می‌تواند توسط یک اعلان رو به جلو از کلاس B جایگزین شود تا از مشکل circular include جلوگیری شود.

```
1  // Fig. 9.25: Withdrawal.h
2  // Definition of class Withdrawal that represents a withdrawal transaction.
3  #ifndef WITHDRAWAL_H
4  #define WITHDRAWAL_H
5
6  class Screen; // forward declaration of class Screen
7  class Keypad; // forward declaration of class Keypad
8  class CashDispenser; // forward declaration of class CashDispenser
9  class BankDatabase; // forward declaration of class BankDatabase
10
11 class Withdrawal
12 {
```

```
13 private:
14    // attributes
15    int accountNumber; // account to withdraw funds from
16    double amount; // amount to withdraw
17
18    // references to associated objects
19    Screen &screen; // reference to ATM's screen
20    Keypad &keypad; // reference to ATM's keypad
21    CashDispenser &cashDispenser; // reference to ATM's cash dispenser
22    BankDatabase &bankDatabase; // reference to the account info database
23 }; // end class Withdrawal
24
25 #endif // WITHDRAWAL_H
```

شکل ۲۵-۹ | استفاده از اعلان‌های روبه جلو بجای دستوردهنده‌های #include.

۵- از عملیات‌های قرار گرفته در بخش سوم شکل ۲۰-۹ برای نوشتن نمونه اولیه تابع برای توابع عضو کلاس استفاده کنید. اگر نوع برگشتی خاصی برای یک عملیات مشخص نکرده‌ایم، تابع عضو را با نوع برگشتی **void** اعلان می‌کنیم. با مراجعه به دیاگرام‌های کلاس در شکل‌های ۲۲-۶ الی ۲۵-۶ می‌توان پارامترهای مورد نیاز را اعلان کرد. برای مثال، با افزودن عملیات سراسری **execute** در کلاس **Withdrawal** که دارای یک لیست پارامتری تهی است، نمونه اولیه (prototype) در خط 15 از شکل ۲۶-۹ بدست می‌آید.

```
1  // Fig. 9.26: Withdrawal.h
2  // Withdrawal class definition. Represents a withdrawal transaction.
3  #ifndef WITHDRAWAL_H
4  #define WITHDRAWAL_H
5
6  class Screen; // forward declaration of class Screen
7  class Keypad; // forward declaration of class Keypad
8  class CashDispenser; // forward declaration of class CashDispenser
9  class BankDatabase; // forward declaration of class BankDatabase
10
11 class Withdrawal : public Transaction
12 {
13 public:
14    // operations
15    void execute(); // perform the transaction
16 private:
17    // attributes
18    int accountNumber; // account to withdraw funds from
19    double amount; // amount to withdraw
20
21    // references to associated objects
22    Screen &screen; // reference to ATM's screen
23    Keypad &keypad; // reference to ATM's keypad
24    CashDispenser &cashDispenser; // reference to ATM's cash dispenser
25    BankDatabase &bankDatabase; // reference to the account info database
26 }; // end class Withdrawal
27
28 #endif // WITHDRAWAL_H
```

شکل ۲۶-۹ | افزودن عملیات‌های به فایل سرآیند کلاس Withdrawal.

با انجام اینکار بحث ما درباره اصول اولیه تولید فایل‌های سرآیند کلاس از روی دیاگرام‌های UML به پایان می‌رسد.

تمرینات خودآزمایی مبحث آموزشی مهندسی نرم‌افزار

۹-۱ تعیین کنید عبارت زیر صحیح است یا اشتباه، در صورت اشتباه بودن، توضیح دهید چرا:

اگر صفتی از یک کلاس با علامت منفی (-) در دیاگرام کلاس نشانه‌گذاری شود، آن صفت بطور مستقیم از خارج از کلاس در دسترس نمی‌تواند باشد.

۹-۲ در شکل ۲۱-۹، رابطه مابین **ATM** و **Screen** براین نکته دلالت دارد که:

a) می‌توانیم از **Screen** به **ATM** هدایت شویم.

b) می‌توانیم از **ATM** به **Screen** هدایت شویم.

c) a و b هر دو صحیح هستند، هدایت دوسویه است.

d) هیچ کدامیک از موارد فوق.

۹-۳ کدی به زبان C++ بنویسید که طرح بکار رفته برای کلاس **Account** را پیاده‌سازی کند.

پاسخ خودآزمایی

۹-۱ صحیح. علامت منفی (-) نشاندهنده رویت private است.

۹-۲ b.

۹-۳ نتیجه طراحی کلاس **Account** در فایل سرآیند شکل ۲۷-۹ آورده شده است.

```
// Fig. 9.27: Account.h
// Account class definition. Represents a bank account.
#ifndef ACCOUNT_H
#define ACCOUNT_H

class Account
{
public:
   Account( int, int, double, double ); // constructor sets attributes
   bool validatePIN( int ) const; // is user-specified PIN correct?
   double getAvailableBalance() const; // returns available balance
   double getTotalBalance() const; // returns total balance
   void credit( double ); // adds an amount to the Account balance
   void debit( double ); // subtracts an amount from the Account balance
   int getAccountNumber() const; // returns account number
private:
   int accountNumber; // account number
   int pin; // PIN for authentication
   double availableBalance; // funds available for withdrawal
   double totalBalance; // funds available + funds waiting to clear
}; // end class Account

#endif // ACCOUNT_H
```

شکل ۲۷-۹ | فایل سرآیند کلاس Account براساس شکل‌های ۲۰-۹ و ۲۱-۹.

خودآزمایی

۹-۱ جاهای خالی را با کلمات مناسب پر کنید:

a) اعضای کلاس از طریق عملگر ..... در ترکیب با نام یک شی از کلاس یا از طریق عملگر..... در ترکیب با اشاره‌گر به یک شی از کلاس در دسترس قرار می‌گیرند.

b) اعضای کلاس بعنوان ..... مشخص می‌شوند و فقط برای توابع عضو کلاس و دوستان کلاس در دسترس قرار می‌گیرند.

c) اعضای کلاس بعنوان..... مشخص می‌شوند و از هر کجای قلمرو کلاس در دسترس می‌گیرند.

d) از ..... می‌توان برای تخصیص یک شی از کلاس به شی از همان کلاس استفاده کرد.

۹-۲ خطا یا خطاهای موجود در هر یک از عبارات زیر را یافته و آنها را اصلاح کنید.

a) فرض کنید نمونه اولیه زیر در کلاس Time اعلان شده‌است:

```
void ~Time(int);
```

b) عبارت زیر بخشی از تعریف کلاس Time است:

```
class Time
{
 public:
       //Function prototype:
 private:
   int hour = ;
   int minute = ;
   int second = ;
} ; // end class Time
```

c) با فرض اینکه نمونه اولیه زیر در کلاس Employee اعلان شده است.

```
int Employee(const char *, const char *);
```

پاسخ خودآزمایی

۹-۱ a) نقطه، b ->) private c) public d) تخصیص memberwise

۹-۲

a) خطا: نابود کننده‌ها اجازه برگشت دادن مقدار یا گرفتن آرگومان را ندارند.

اصلاح: حذف نوع برگشتی void و پارامتر int از اعلان.

b) خطا: اعضا نمی‌توانند بصورت صریح در تعریف کلاس مقداردهی اولیه شوند.

اصلاح: حذف مقداردهی صریح از تعریف کلاس و مقداردهی اولیه اعضای داده در یک سازنده.

c) خطا: سازنده‌ها قادر به برگشت دادن مقدار نیستند.

اصلاح : حذف نوع برگشتی int از اعلان.

تمرینات

۹-۳ منظور از عملگر تفکیک قلمرو چیست؟

۹-۴ سازنده‌ای تدارک ببینید که قادر به استفاده از زمان جاری از تابع time( ) باشد، اعلان شده در کتابخانه استاندارد C++ با سرآیند <ctime>، تا یک شی از کلاس time را مقداردهی اولیه نماید.

۹-۵ کلاسی بنام Complex ایجاد کنید که قادر به کار با مقادیر مختلط باشد. برنامه‌ای برای تست کلاس خود بنویسید. اعداد مختلط بفرم زیر هستند

realPart + imaginaryPart * i (بخش موهومی + بخش حقیقی * i)

که در آن i برابر است با $\sqrt{-1}$

از متغیرهای double برای عرضه داده private کلاس استفاده کنید. یک سازنده در نظر بگیرید که به شی از این کلاس امکان مقداردهی اولیه را در زمان اعلان فراهم آورد. باید سازنده حاوی مقادیر پیش فرض باشد. توابع عضو public را در نظر بگیرید که وظایف زیر را انجام دهند:

a) جمع دو عدد complex : بخش‌های حقیقی با یکدیگر و بخش‌های موهومی با یکدیگر جمع می‌شوند.

b) تفریق دو عدد complex : بخش حقیقی قرار گرفته در سمت راست تفریق از بخش حقیقی قرار گرفته در سمت چپ عملوند، کاسته می‌شود، و بخش موهومی قرار گرفته در سمت راست عملوند از بخش موهومی قرار گرفته در سمت چپ عملوند کاسته می‌شود.

c) چاپ اعداد complex بفرم (a,b) که در آن a بخش حقیقی و b بخش موهومی است.

۶-۹ کلاسی بنام Rational ایجاد کنید تا عملیات ریاضی را با کسرها انجام دهد. برنامه‌ای برای تست کلاس بنویسید. از متغیرهای صحیح برای عرضه داده private کلاس، numerator و denominator استفاده کنید. یک سازنده در نظر بگیرید که به شی از این کلاس امکان مقداردهی اولیه را در زمان اعلان فراهم آورد. سازنده باید حاوی مقادیر پیش فرض باشد و باید کسر را بفرم کاسته شده ذخیره کند. برای مثال، کسر 2/4 می‌تواند در یک شی بصورت 1 در numerator و 2 در denominator ذخیره شود. توابع عضو public را برای انجام وظایف زیر در نظر بگیرید:

a) جمع دو عدد Rational. نتیجه باید بفرم کاسته شده ذخیره شود.

b) تفریق دو عدد Rational. نتیجه باید بفرم کاسته شده ذخیره شود.

c) ضرب دو عدد Rational. نتیجه باید بفرم کاسته شده ذخیره شود.

d) تقسیم دو عدد Rational. نتیجه باید بفرم کاسته شده ذخیره شود.

e) چاپ اعداد Rational. بفرم a/b ، که در آن a صورت و b مخرج کسر است.

f) چاپ اعداد Rational. با فرمت اعشاری.

۷-۹ برنامه شکل‌های ۸-۹ و ۹-۹ را به نحوی اصلاح کنید که حاوی تابع عضو tick باشد تا زمان ذخیره شده درشی Time را در هر ثانیه افزایش دهد. بایستی شی time همیشه در وضعیت پایدار باقی بماند. برنامه‌ای بنویسید که تابع عضو tick را در حلقه‌ای که زمان را در فرمت استاندارد در هر بار تکرار چاپ می‌کند، تست نماید تا از عملکرد صحیح آن مطمئن گردیم . حتماً حالات زیر تست شوند:

a) ورود به دقیقه بعد.

b) ورود به ساعت بعد.

c) ورود به روز بعد (یعنی 11:59:59 PM به 12:00:00AM).

۸-۹ کلاس Date بکار رفته در شکل‌های ۱۷-۹ و ۱۸-۹ را برای انجام تست خطا در مقداردهی مقادیر برای اعضای داده month ، day و year تغییر دهید. همچنین تابع عضو nextDay را برای افزایش یک روز در هر بار در نظر بگیرید. شی Date باید همیشه در وضعیت پایدار باقی بماند. برنامه‌ای بنویسید که تابع nextDay را در حلقه‌ای که

تاریخ جاری را در هر بار تکرار چاپ می‌کند، تست نماید تا از عملکرد صحیح nextDay مطمئن گردیم. حتماً حالات زیر تست شوند:

a) ورود به ماه بعد.

b) ورود به سال بعد.

۹-۹ کلاس اصلاح شده Time در تمرین ۷-۹ و کلاس اصلاح شده Date در تمرین ۸-۹ را بصورت یک کلاس بنام DateAndTime با هم ترکیب کنید. اگر زمان برای ورود به روز بعدی افزایش یابد، تابع tick را برای فراخوانی تابع nextDay اصلاح کنید. توابع printStandard , PrintUniversal را برای چاپ تاریخ و زمان اصلاح کنید. برنامه‌ای بنویسید که کلاس جدید DateAndTime را تست کند.

۱۰-۹ توابع set بکار رفته در کلاس Time شکل‌های ۸-۹ و ۹-۹ را به نحوی اصلاح کنید تا در صورتیکه مبادرت به مقداردهی یک شی از کلاس Time با مقدار نامعتبر شود، خطای متناسب با آن برگشت داده شود. برنامه‌ای برای تست این نسخه از کلاس بنویسید. پیغام‌های خطا را در صورت برگشت مقادیر خطا از توابع set بنمایش درآورید.

۱۱-۹ کلاس بنام Rectangle با صفات width، length ایجاد کنید که هر یک دارای مقدار پیش فرض 1 هستند. توابع عضوی در نظر بگیرید که اقدام به محاسبه مساحت و محیط مستطیل کنند. همچنین توابع get و set را برای صفات length, width در نظر بگیرید. بایستی توابع set مراقب باشند که length, width دارای مقادیر اعشاری بزرگتر از 0.0 و کمتر از 20.0 باشند.

۱۲-۹ کلاس پیشرفته Rectangle را به نسبت کلاس یاد شده در تمرین ۱۱-۹ ایجاد کنید. این کلاس فقط مختصصات دکارت را برای چهار گوشه مستطیل ذخیره می‌کند. سازنده مبادرت به فراخوانی یک تابع set می‌کند که مجموعه چهار مختصصاتی را پذیرفته و بررسی می‌کند که هر کدام از آنها در اولین ربع قرار دارند و مقدار آنها بزرگتر از 0.0 نمی‌باشد. همچنین تابع set بررسی می‌کند که مختصات تدارک دیده شده، خاص یک مستطیل باشند. توابع عضو در نظر بگیرید که مبادرت به محاسبه طول، عرض، محیط و مساحت نمایند. همچنین یک تابع مسند بنام square در نظر بگیرید که تعیین کنید آیا با مستطیل طرف هستیم یا مربع.

۱۳-۹ کلاس Rectangle مطرح شده در تمرین ۱۲-۹ را برای داشتن تابع draw اصلاح کنید. این تابع اقدام به نمایش مستطیل در درون یک جعبه 25 در 25 می‌کند که بخشی از ربع اول مستطیل در آن مقیم است. از تابع setFillCharacter برای پر کردن داخل مستطیل با کاراکتر مشخص شده استفاده کنید. همچنین از تابعی بنام setPerimeterCharacter برای تعیین کاراکتری که از آن برای رسم بدنه یا حاشیه مستطیل استفاده خواهد شد، کمک بگیرید.

۱۴-۹ کلاس بنام HugeInteger ایجاد کنید که از یک آرایه 40 عنصری از ارقام برای ذخیره‌سازی ارقام به بزرگی اعداد 40 رقمی استفاده کند. توابع عضو substract , add, output, input را در نظر بگیرید. برای مقایسه شی‌های HugeInteger، توابع isEqualTo، isNotEqualTo، isGreaterThan، isLessThan, isGreaterThanOrEqualTo و isLessThanOrEqualTo را در نظر بگیرید. هر یک از این توابع یک تابع پیشگو یا مسند هستند که در صورت

برقرار بودن رابطه مابین دو شی IntegerHuge مقدار true و در صورت برقرار نبودن رابطه مقدار false برگشت می‌دهند. همچنین تابع مسند isZero را در نظر بگیرید. در صورت تمایل می‌توانید توابع عضو ,divide , modulus multiply را هم بکار گیرید.

فصل دهم

کلاس‌ها: نگاهی عمیق‌تر: بخش II

اهداف

در این فصل با مطالب زیر آشنا خواهید شد:

- مشخص کردن شی‌های ثابت (const) و توابع عضو ثابت.
- ایجاد شی‌های مرکب از سایر شی‌ها.
- استفاده از توابع و کلاس‌های friend.
- استفاده از اشاره‌گر this.
- ایجاد و نابود کردن شی‌های دینامیکی با عملگر new و delete.
- استفاده از اعضای داده static و توابع عضو.
- نکاتی در ارتباط با کلاس‌های تکرار شونده که در میان عناصر کلاس‌های حامل حرکت می‌کنند.
- استفاده از کلاس‌های پروکسی برای پنهان نگه‌داشتن جزئیات پیاده‌سازی از دید کلاس‌های سرویس‌گیرنده.

رئوس مطالب

۱۰-۱ مقدمه

در این فصل به آموزش کلاس‌ها و داده‌های انتزاعی به همراه چندین مبحث پیشرفته ادامه می‌دهیم. از شی‌ها و توابع عضو **const** برای جلوگیری کردن از تغییر شی‌ها و حفظ حداقل مجوزهای دسترسی استفاده خواهیم کرد. در مورد ترکیب صحبت می‌کنیم که فرمی از استفاده مجدد است که در آن کلاسی می‌تواند دارای شی‌های از سایر کلاس‌ها بعنوان اعضا باشد. سپس به معرفی مبحث دوستی (*friendshisp*) می‌پردازیم، که به طراح کلاس امکان می‌دهد تا توابع غیرعضوی را که می‌توانند به اعضای غیرسراسری (*public*) کلاس دسترسی پیدا کنند را معین نماید. تکنیکی که غالباً در سربارگذاری عملگر بکار گرفته می‌شود (فصل یازدهم). در مورد یک اشاره‌گر خاص بنام **this** صحبت می‌کنیم که یک آرگومان ضمنی برای هر تابع عضو غیراستاتیک کلاس است که به این توابع عضو اجازه دسترسی صحیح به اعضاء داده شی و سایر توابع عضو غیراستاتیکی را فراهم می‌آورد. سپس در ارتباط با مدیریت حافظه دینامیکی صحبت می‌کنیم و نشان می‌دهیم که چگونه با استفاده از عملگرهای **new** و **delete** می‌توان شی‌های دینامیکی را ایجاد و نابود کرد. سپس به بررسی اعضای کلاس استاتیک و نحوه استفاده از اعضای داده استاتیک و توابع عضو در کلاس‌های متعلق بخودمان می‌پردازیم. در پایان، با نحوه ایجاد یک کلاس پروکسی برای پنهان کردن جزئیات پیاده‌سازی یک کلاس (شامل اعضای داده **private** آن) از دید سرویس‌گیرنده‌های کلاس آشنا خواهید شد.

در فصل سوم به معرفی کلاس استاندارد **string** پرداختیم. با این وجود، در این فصل از رشته‌های مبتنی بر اشاره‌گر استفاده خواهیم کرد که در فصل هشتم معرفی شده است تا کسانی که مایل هستند تا خود را آماده کارهای حرفه‌ای نمایند، از آن کمک بگیرند.

۲-۱۰ شی‌های ثابت (const) و توابع عضو ثابت

یکی از قواعد بنیادین در مهندسی نرم‌افزار، حفظ حداقل مجوزها و پایبندی به آنها است. اجازه دهید تا به بررسی این قواعد در ارتباط با شی‌ها بپردازیم. برخی از شی‌ها نیاز به اصلاح و تغییر دارند و تعدادی هم ندارند. برنامه‌نویس می‌تواند با استفاده از کلمه کلیدی **const** مشخص کند که یک شی تغییرپذیر نبوده و هر عملی که منجر به تغییر آن شی شود با خطای کامپایل مواجه می‌شود. عبارت

```
const Time noon (12,0,0);
```

شی **noon** از کلاس **Time** را بصورت ثابت (**const**) اعلان و با 12 ظهر مقداردهی اولیه کرده است.

مهندسی نرم‌افزار

اعلان یک شی بعنوان ثابت، سبب حفظ حداقل مجوزها یا امتیازها می‌شود.

کارائی

اعلان متغیرها و شی‌ها بصورت ثابت می‌تواند در افزایش کارایی نقش داشته باشد.

امروزه برخی از کامپایلرهای پیشرفته قادر به انجام بهینه‌سازی‌های مشخص بر روی ثابت‌ها هستند که نمی‌توان بر روی متغیرها اعمال کرد. کامپایلرهای C++ اجازه فراخوانی تابع عضو برای شی‌های ثابت را نمی‌دهند مگر اینکه خود توابع عضو بصورت ثابت اعلان شده باشند. این امر حتی در مورد توابع عضو *get* هم که شی را دچار تغییر نمی‌سازند صادق است. علاوه بر این، کامپایلر به توابع عضو اعلان شده بصورت **const** اجازه تغییر در شی را نمی‌دهد.

تابعی بصورت ثابت اعلان می‌شود که هم در نمونه اولیه خود (شکل۱-۱۰، خطوط 24-19) و هم در تعریف خود (شکل۲-۱۰، خطوط 47، 53، 59 و 65) با اعمال کلمه‌کلیدی **const** پس از لیست پارامتری تابع و قبل از براکت چپ که شروع بدنه تابع است (در مورد تعریف تابع) مشخص شده باشد.

خطای برنامه‌نویسی

تعریف یک تابع عضو ثابت که اقدام به تغییر در عضو داده یک شی می‌نماید، خطای کامپایل است.

خطای برنامه‌نویسی

تعریف یک تابع عضو ثابت که یک تابع عضو غیرثابت از کلاسی که از همان کلاس ساخته شده است، خطای کامپایل بدنبال خواهد داشت.

خطای برنامه‌نویسی

فعال کردن یک تابع عضو غیرثابت بر روی یک شی ثابت، خطای کامپایل است.

در این بین برای سازنده‌ها و نابودکننده‌ها که غالباً مبادرت به تغییر در شی‌ها می‌کنند، مشکل بوجود می‌آید. اعلان **const** اجازه‌ای برای سازنده‌ها و نابودکننده‌ها نمی‌دهد. یک سازنده بایستی اجازه تغییر در یک شی را داشته باشد از اینروست که شی می‌تواند بدرستی مقداردهی اولیه شود. یک نابودکننده باید قادر به انجام عملیات قبل از خاتمه قبل از اینکه حافظه اخذ شده توسط شی از سوی برنامه بازپس گرفته شده باشد.

خطای برنامه‌نویسی

اقدام به اعلان یک سازنده یا نابودکننده ***const****، خطای کامپایل است.*

تعریف و استفاده از توابع عضو ثابت

در برنامه شکل‌های ۱-۱۰ الی ۳-۱۰ کلاس **Time** از برنامه‌های ۹-۹ و ۱۰-۹ با اعمال توابع *get* و تابع ثابت **printUniversal** اصلاح شده است. در فایل سرآیند **Time.h** (شکل ۱-۱۰)، خطوط 21-19 و 24 حاوی کلمه‌کلیدی **const** پس از هر لیست پارامتری تابع هستند. تعریف متناظر با هر تابع در شکل ۲-۱۰ آورده شده است (خطوط 59، 53، 47 و 65) با اعمال کلمه‌کلیدی **const** پس لیست پارامتری هر تابع. در شکل ۳-۱۰ دو نمونه از شی **Time** ایجاد شده است. شی **wakeUp** بصورت غیرثابت (خط 7) و شی **noon** بصورت ثابت (خط 8). برنامه اقدام به فعال کردن توابع عضو غیرثابت **setHour** (خط 13) و **printStandard** (خط 20) بر روی شی ثابت **noon** می‌کند. در هر مورد، کامپایلر یک پیغام خطا تولید می‌کند. همچنین برنامه فراخوانی سه تابع عضو دیگر را عرضه کرده است. یک تابع عضو غیرثابت بر روی یک شی غیرثابت (خط 11)، یک تابع عضو ثابت بر روی یک شی غیرثابت (خط 15) و یک تابع عضو ثابت بر روی یک شی ثابت (خطوط 18-17). پیغام‌های خطای تولید شده از فراخوانی توابع عضو غیرثابت بر روی یک شی ثابت در خروجی برنامه نشان داده شده‌اند. توجه کنید که، اگرچه برخی از کامپایلرهای جاری فقط پیغام هشدار برای خطوط 13 و 20 صادر می‌کنند (که در اینحالت برنامه اجرا می‌شود)، اما ما به این هشدار بعنوان خطا نگاه می‌کنیم. استاندارد ANSI/ISO C++ اجازه فراخوانی یک تابع عضو غیرثابت بر روی یک شی ثابت را نمی‌دهد.

```
1  // Fig. 10.1: Time.h
2  // Definition of class Time.
3  // Member functions defined in Time.cpp.
4  #ifndef TIME_H
5  #define TIME_H
6
7  class Time
8  {
9  public:
10    Time( int = 0, int = 0, int = 0 ); // default constructor
11
12    // set functions
13    void setTime( int, int, int ); // set time
14    void setHour( int ); // set hour
15    void setMinute( int ); // set minute
16    void setSecond( int ); // set second
17
```

```
18    // get functions (normally declared const)
19    int getHour() const; // return hour
20    int getMinute() const; // return minute
21    int getSecond() const; // return second
22
23    // print functions (normally declared const)
24    void printUniversal() const; // print universal time
25    void printStandard(); // print standard time (should be const)
26 private:
27    int hour; // 0 - 23 (24-hour clock format)
28    int minute; // 0 - 59
29    int second; // 0 - 59
30 }; // end class Time
31
32 #endif
```

شکل ۱۰-۱ | تعریف کلاس Time با توابع عضو const.

```
1  // Fig. 10.2: Time.cpp
2  // Member-function definitions for class Time.
3  #include <iostream>
4  using std::cout;
5
6  #include <iomanip>
7  using std::setfill;
8  using std::setw;
9
10 #include "Time.h" // include definition of class Time
11
12 // constructor function to initialize private data;
13 // calls member function setTime to set variables;
14 // default values are 0 (see class definition)
15 Time::Time( int hour, int minute, int second )
16 {
17    setTime( hour, minute, second );
18 } // end Time constructor
19
20 // set hour, minute and second values
21 void Time::setTime( int hour, int minute, int second )
22 {
23    setHour( hour );
24    setMinute( minute );
25    setSecond( second );
26 } // end function setTime
27
28 // set hour value
29 void Time::setHour( int h )
30 {
31    hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
32 } // end function setHour
33
34 // set minute value
35 void Time::setMinute( int m )
36 {
37    minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
38 } // end function setMinute
39
40 // set second value
41 void Time::setSecond( int s )
42 {
43    second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
44 } // end function setSecond
45
46 // return hour value
47 int Time::getHour() const // get functions should be const
48 {
49    return hour;
50 } // end function getHour
51
```

```
52 // return minute value
53 int Time::getMinute() const
54 {
55    return minute;
56 } // end function getMinute
57
58 // return second value
59 int Time::getSecond() const
60 {
61    return second;
62 } // end function getSecond
63
64 // print Time in universal-time format (HH:MM:SS)
65 void Time::printUniversal() const
66 {
67    cout << setfill( '0' ) << setw( 2 ) << hour << ":"
68       << setw( 2 ) << minute << ":" << setw( 2 ) << second;
69 } // end function printUniversal
70
71 // print Time in standard-time format (HH:MM:SS AM or PM)
72 void Time::printStandard() // note lack of const declaration
73 {
74    cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
75       << ":" << setfill( '0' ) << setw( 2 ) << minute
76       << ":" << setw( 2 ) << second << ( hour < 12 ? " AM" : " PM" );
77 } // end function printStandard
```

شکل ۲-۱۰ | تعریف تابع عضو کلاس Time، شامل توابع عضو ثابت.

```
1  // Fig. 10.3: fig10_03.cpp
2  // Attempting to access a const object with non-const member functions.
3  #include "Time.h" // include Time class definition
4
5  int main()
6  {
7     Time wakeUp( 6, 45, 0 ); // non-constant object
8     const Time noon( 12, 0, 0 ); // constant object
9
10                              // OBJECT      MEMBER FUNCTION
11    wakeUp.setHour( 18 );     // non-const   non-const
12
13    noon.setHour( 12 );       // const       non-const
14
15    wakeUp.getHour();         // non-const   const
16
17    noon.getMinute();         // const       const
18    noon.printUniversal();    // const       const
19
20    noon.printStandard();     // const       non-const
21    return 0;
22 } // end main
```

Borland C++ command-line compiler error messages

```
Warning W8037 fig10_03.ccp 13:Non-const function Time::setHour(int)
     called for const object in function main()
Warning W8037 fig10_03.ccp 20:Non-const function Time::printStandard()
     called for const object in function main()
```

Microsoft Visual C++.NET compiler error message

```
C:\cpphtp5_examples\ch10\Fig10_01_03\fig10_03.cpp(13) : error C2662:
   'Time::setHour' : cannot convert 'this' pointer from 'const Time' to
   'Time &'
       Conversion loses qualifiers
C:\cpphtp5_examples\ch10\Fig10_01_03\fig10_03.cpp(20) : error C2662:
   'Time::printStandard':cannot convert 'this' pointer from 'const Time'
to
   'Time &'
       Conversion loses qualifiers
```

GNU C++ compiler error message

```
fig10_03.ccp:13: error: passing 'const Time' as 'this' argument of
```

```
     'void Time::setHour(int)' discards qualifiers
fig10_03.ccp:20: error: passing 'const Time' as 'this' argument of
     'void Time::printStandard()' discards qualifiers
```

شکل ۳-۱۰ | شی‌های ثابت و توابع عضو ثابت.

دقت کنید که حتی اگر یک سازنده بصورت یک تابع عضو غیرثابت باشد (شکل ۲-۱۰، خطوط 18-15) هنوز هم می‌تواند در مقداردهی اولیه یک شی **const** بکار گرفته شود (شکل ۳-۱۰، خط 8). تعریف سازنده Time (شکل ۲-۱۰ خطوط 18-15) نشان می‌دهد که سازنده **Time** تابع عضو غیرثابت دیگری بنام **setTime** (خطوط 26-21) را برای انجام مقداردهی اولیه یک شی **Time** فراخوانی می‌کند. فراخوانی یک تابع عضو غیرثابت از طریق فراخوانی سازنده بعنوان بخشی از مقداردهی اولیه یک شی ثابت، امکان‌پذیر است. توجه کنید که در خط 20 از شکل ۳-۱۰ یک خطای کامپایل تولید می‌شود حتی اگر تابع عضو **printStandard** از کلاس **Time** اقدام به تغییر در شی نکند.

مقداردهی اولیه یک عضو داده ثابت با یک مقداردهی کننده عضو

در برنامه شکل‌های ۴-۱۰ الی ۶-۱۰ به معرفی روش استفاده از گرامر مقداردهی کننده عضو می‌پردازیم. تمام اعضای داده می‌توانند با استفاده از گرامر مقداردهی کننده عضو، مقداردهی اولیه شوند، اما اعضای داده ثابت و اعضای داده که مورد مراجعه هستند بایستی با استفاده از مقداردهی کننده‌های عضو مقداردهی اولیه شوند. در ادامه این فصل، خواهید دید که شی‌های عضو بایستی به این روش مقداردهی اولیه شوند. در فصل دوازدهم به هنگام آموزش توارث، شاهد خواهید بود که قسمت‌های مبتنی بر کلاس از کلاس‌های مشتق شده هم بایستی به این روش مقداردهی اولیه شوند.

تعریف سازنده (شکل ۵-۱۰، خطوط 16-11) از لیست مقداردهی کننده عضو برای مقداردهی اولیه اعضای داده کلاس **Increment** بنام **count** که از نوع صحیح و ثابت نیست و **increment** از نوع صحیح و ثابت است (اعلان شده در خطوط 20-19 از شکل ۴-۱۰) استفاده کرده است. مقداردهی کننده عضو مابین یک لیست پارامتری سازنده و براکت چپ ظاهر می‌شوند که بدنه سازنده با آن آغاز می‌شود. لیست مقداردهی کننده عضو (شکل ۵-۱۰، خطوط 13-12) از لیست پارامتری توسط یک کولن (:) جدا شده است. هر مقداردهی کننده عضو متشکل از نام داده عضو بدنبال آن پرانتزهای حاوی مقدار اولیه عضو است. در این مثال، **count** با مقدار پارامتر سازنده **c** و **increment** با مقدار پارامتر سازنده **i** مقداردهی شده است. توجه کنید که مقداردهی کننده‌های عضو مضاعف توسط کاما از یکدیگر جدا می‌شوند. همچنین لیست مقداردهی کننده عضو قبل از اینکه بدنه سازنده اجرا شود، اجرا می‌شود.

```
1  // Fig. 10.4: Increment.h
2  // Definition of class Increment.
3  #ifndef INCREMENT_H
4  #define INCREMENT_H
5
6  class Increment
7  {
```

```
8  public:
9     Increment( int c = 0, int i = 1 ); // default constructor
10
11    // function addIncrement definition
12    void addIncrement()
13    {
14       count += increment;
15    } // end function addIncrement
16
17    void print() const; // prints count and increment
18 private:
19    int count;
20    const int increment; // const data member
21 }; // end class Increment
22
23 #endif
```

شکل ٤-١٠ | تعریف کلاس Increment حاوی عضو داده غیرثابت count و عضو داده ثابت increment.

```
1  // Fig. 10.5: Increment.cpp
2  // Member-function definitions for class Increment demonstrate using a
3  // member initializer to initialize a constant of a built-in data type.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  #include "Increment.h" // include definition of class Increment
9
10 // constructor
11 Increment::Increment( int c, int i )
12    : count( c ), // initializer for non-const member
13      increment( i ) // required initializer for const member
14 {
15    // empty body
16 } // end constructor Increment
17
18 // print count and increment values
19 void Increment::print() const
20 {
21    cout << "count = " << count << ", increment = " << increment << endl;
22 } // end function print
```

شکل ٥-١٠ | استفاده از مقداردهی کننده عضو برای مقداردهی یک ثابت از نوع توکار.

```
1  // Fig. 10.6: fig10_06.cpp
2  // Program to test class Increment.
3  #include <iostream>
4  using std::cout;
5
6  #include "Increment.h" // include definition of class Increment
7
8  int main()
9  {
10    Increment value( 10, 5 );
11
12    cout << "Before incrementing: ";
13    value.print();
14
15    for ( int j = 1; j <= 3; j++ )
16    {
17       value.addIncrement();
18       cout << "After increment " << j << ": ";
19       value.print();
20    } // end for
21
22    return 0;
23 } // end main
```

```
Before incrementing: count = 10, increment = 5
After increment 1: count = 15, increment = 5
After increment 2: count = 20, increment = 5
```

```
After increment 3: count = 25, increment = 5
```

شکل ٦-١٠ | فراخوانی توابع عضو print و addIncrement از شی increment.

مقداردهی اشتباه یک عضو داده ثابت با عبارت تخصیصی

در برنامه شکل‌های ۷-۱۰ الی ۹-۱۰ به توضیح خطاهای کامپایل رخ داده به هنگام مبادرت به مقداردهی عضو داده ثابت **increment** با یک عبارت تخصیصی (شکل ۸-۱۰، خط 14) در بدنه سازنده **Increment** بجای یک لیست مقداردهی کننده عضو پرداخته شده است. به خط 13 از شکل ۸-۱۰ توجه کنید که خطای کامپایل تولید نمی‌کند، چرا که **count** بصورت ثابت (**const**) اعلان نشده است. همچنین به خطاهای کامپایل تولید شده توسط Microsoft Visual C++.NET در اشاره به عضو داده **increment** از نوع **int** بعنوان یک «*شی ثابت*» توجه کنید. همانند کلاس‌های نمونه‌سازی شده، متغیرهای که از نوع‌های بنیادین هستند هم در حافظه فضا اشغال می‌کنند و از اینرو غالباً از آنها بعنوان «شی» یاد می‌شود.

خطای برنامه‌نویسی

لیست مقداردهی کننده عضو برای یک عضو داده ثابت فراهم نکنید که با خطای کامپایل مواجه می‌شوید.

مهندسی نرم‌افزار

اعضای داده ثابت (شی‌های ثابت و متغیرهای ثابت) و اعضای داده اعلان شده بعنوان مراجعه باید با گرامر مقداردهی کننده عضو، مقداردهی اولیه شوند، اقدام به تخصیص به این نوع از داده‌ها در بدنه سازنده مجاز نمی‌باشد.

توجه کنید که تابع **print** (شکل ۸-۱۰، خطوط 21-18) بصورت ثابت اعلان شده است. ممکن است این عنوان برای این تابع کمی عجیب بنظر برسد، چرا که محتملاً برنامه هرگز دارای یک شی **Increment** ثابت نخواهد بود. با این وجود، ممکن است که برنامه یک مراجعه ثابت به یک شی **Increment** یا یک اشاره‌گر به ثابتی داشته باشد که به یک شی **Increment** اشاره می‌کند. معمولاً اینحالت زمانی رخ می‌دهد که شی‌های از کلاس **Increment** به توابع ارسال یا از توابع برگشت داده شوند. در این موارد، فقط توابع عضو ثابت از کلاس **Increment** می‌توانند از طریق مراجعه یا اشاره‌گر فراخوانی شوند. بنابر این، اعلان تابع **print** بصورت ثابت، معقول بنظر می‌رسد. با انجام چنین کاری از رخ دادن خطاهای در این رابطه جلوگیری می‌شود.

اجتناب از خطا

*تمام توابع عضو کلاس را که در ناحیه عملکردی خود مبادرت به اعمال تغییر در شی نمی‌کنند، از نوع **const**(ثابت) اعلان کنید.*

```
1  // Fig. 10.7: Increment.h
2  // Definition of class Increment.
3  #ifndef INCREMENT_H
4  #define INCREMENT_H
5
```

```
6  class Increment
7  {
8  public:
9     Increment( int c = 0, int i = 1 ); // default constructor
10
11    // function addIncrement definition
12    void addIncrement()
13    {
14       count += increment;
15    } // end function addIncrement
16
17    void print() const; // prints count and increment
18 private:
19    int count;
20    const int increment; // const data member
21 }; // end class Increment
22
23 #endif
```

شکل ۷-۱۰ | تعریف کلاس Increment حاوی یک عضو داده غیرثابت counst و عضو داده ثابت increment.

```
1  // Fig. 10.8: Increment.cpp
2  // Attempting to initialize a constant of
3  // a built-in data type with an assignment.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  #include "Increment.h" // include definition of class Increment
9
10 // constructor; constant member 'increment' is not initialized
11 Increment::Increment( int c, int i )
12 {
13    count = c; // allowed because count is not constant
14    increment = i; // ERROR: Cannot modify a const object
15 } // end constructor Increment
16
17 // print count and increment values
18 void Increment::print() const
19 {
20    cout << "count = " << count << ", increment = " << increment << endl;
21 } // end function print
```

شکل ۸-۱۰ | مقداردهی سهوی یک ثابت از نوع توکار توسط عبارت تخصیصی.

```
1  // Fig. 10.9: fig10_09.cpp
2  // Program to test class Increment.
3  #include <iostream>
4  using std::cout;
5
6  #include "Increment.h" // include definition of class Increment
7
8  int main()
9  {
10    Increment value( 10, 5 );
11
12    cout << "Before incrementing: ";
13    value.print();
14
15    for ( int j = 10; j <= 3; j++ )
16    {
17       value.addIncrement();
18       cout << "After increment " << j << ": ";
19       value.print();
20    } // end for
21
22    return 0;
23 } // end main
```

Borland C++ command-line compiler error messages

```
Error E2024 Increment.cpp 14: Cannot modify a const object in function
```

```
    Increment::Increment(int,int)
```

Microsoft Visual C++.NET compiler error message

```
C:\cpphtp5_examples\ch10\Fig10_07_09\Increment.cpp(12) : error C2758:
   'Increment::increment' :must be initialized in constructor
   base/member initializer list
        C:\cpphtp5_examples\ch10\Fig10_07_09\Increment.h(20) :
        see declaration of 'Increment::increment'
C:\cpphtp5_examples\ch10\Fig10_07_09\Increment.cpp(14): error C2166:
   l-value specifies const object
```

GNU C++ compiler error message

```
Increment.cpp:12: error: uninitialized member 'Increment::increment'with
     'const'type 'const int'
Increment.cpp:14: error: assigment of read-only data-member
     'Increment::increment'
```

شکل ۹-۱۰ | برنامه تست‌کننده کلاس Increment که خطاهای کامپایل تولید می‌کند.

۳-۱۰ ترکیب: شی‌ها بعنوان اعضای کلاس

یک شی **AlarmClock** نیاز دارد تا از زمان فرض شده برای بصدا در آوردن زنگ مطلع باشد، پس چرا نبایستی یک شی **Time** بعنوان عضوی از کلاس **AlarmClock** بحساب آورده نشود؟ چنین قابلیتی *ترکیب* نامیده می‌شود و گاهی اوقات بعنوان «بستگی داشتن یا رابطه داشتن» شناخته می‌شود. یک کلاس می‌تواند شی‌های از سایر کلاس‌ها را بعنوان اعضاء داشته باشد.

مهندسی نرم‌افزار

یکی از فرم‌های استفاده مجدد از نرم‌افزار، ترکیب است که در آن یک کلاس دارای شی‌های از سایر کلاس‌ها بعنوان اعضا است.

زمانیکه یک شی ایجاد می‌شود، سازنده آن بصورت اتوماتیک فراخوانی می‌گردد. قبلاً شاهد نحوه ارسال آرگومان‌ها به سازنده یک شی که در **main** ایجاد می‌کردیم بوده‌اید. در این بخش شاهد خواهید بود که چگونه سازنده یک شی می‌تواند آرگومان‌ای به سازنده‌های عضو شی ارسال کند که از طریق از مقداردهی کننده‌های عضو صورت می‌گیرد. شی‌های عضو به ترتیبی که در تعریف کلاس اعلان شده‌اند (نه به ترتیبی که در لیست مقداردهی کننده عضو سازنده ظاهر شده‌اند) و قبل از ایجاد شی‌های کلاس احاطه کننده (شی‌های میزبان) ساخته می‌شوند.

در برنامه شکل‌های ۱۰-۱۰ الی ۱۴-۱۰ از کلاس **Date** (شکل‌های ۱۰-۱۰ و ۱۱-۱۰) و کلاس **Employee** (شکل‌های ۱۲-۱۰ و ۱۳-۱۰) برای نشان دادن شی‌هایی بعنوان اعضای سایر شی‌ها استفاده شده است. تعریف کلاس **Employee** (شکل ۱۲-۱۰) حاوی اعضای داده خصوصی بنام‌های **firstName**، **lastName**، **birthDate** و **hireDate** است. اعضای **birthDate** و **hireDate** شی‌های ثابت از کلاس **Date** هستند که حاوی اعضای داده خصوصی بنام‌های **month**، **day** و **year** می‌باشند. سرآیند سازنده **Employee** (شکل ۱۳-۱۰ خطوط 21-18) مشخص می‌کند که سازنده چهار پارامتر دریافت می‌کند(**first**, **last**, **dateOfBirth**, **dateOfHire**). از دو پارامتر اول در بدنه سازنده برای مقداردهی

اولیه آرایه‌های کاراکتری **firstName** و **lastName** استفاده می‌شود. دو پارامتر آخر از طریق لیست‌های مقداردهی کننده به سازنده کلاس **Date** ارسال می‌شوند. کولن (:) بکار رفته در سرآیند منجر به جداسازی مقداردهی کننده‌های عضو از لیست پارامتری می‌شود. مقداردهی کننده‌های عضو، مشخص می‌کنند که پارامترهای سازنده **Employee** به سازنده‌های شی‌های عضو **Date** ارسال می‌گردند. پارامتر **dateOfBirth** به شی سازنده **birthDate** (شکل ۱۳-۱۰، خط 20) و پارامتر **dateOfHire** به شی سازنده **hireDate** (شکل ۱۳-۱۰، خط 21) ارسال می‌شوند. مجموعاً، مقداردهی کننده‌های عضو توسط کاما از یکدیگر جدا شده‌اند.

به هنگام معرفی کلاس **Date** (شکل ۱۰-۱۰) توجه کنید که این کلاس دارای یک سازنده که پارامتری از نوع **Date** دریافت کند نیست. از اینرو چگونه لیست مقداردهی کننده عضو در سازنده کلاس **Employee** قادر به مقداردهی شی‌های **birthDate** و **hireDate** با ارسال شی‌های **Date** به سازنده‌های **Date** آنها صورت می‌گیرد؟ همانطوری که در فصل نهم گفته شد، کامپایلر برای هر کلاس یک سازنده پیش‌فرض کپی کننده که مبادرت به کپی هر عضو از شی از آرگومان شی سازنده به عضو متناظر از شی که مقداردهی می‌شود، می‌کند. در فصل یازدهم خواهید آموخت که چگونه برنامه‌نویسان می‌توانند سازنده‌های کپی کننده بهینه شده تعریف کنند.

```
1  // Fig. 10.10: Date.h
2  // Date class definition; Member functions defined in Date.cpp
3  #ifndef DATE_H
4  #define DATE_H
5
6  class Date
7  {
8  public:
9     Date( int = 1, int = 1, int = 1900 ); // default constructor
10    void print() const; // print date in month/day/year format
11    ~Date(); // provided to confirm destruction order
12 private:
13    int month; // 1-12 (January-December)
14    int day; // 1-31 based on month
15    int year; // any year
16
17    // utility function to check if day is proper for month and year
18    int checkDay( int ) const;
19 }; // end class Date
20
21 #endif
```

شکل ۱۰-۱۰ | تعریف کلاس Date.

در برنامه شکل ۱۴-۱۰ دو شی **Date** ایجاد (خطوط 12-11) و آنها را بعنوان آرگومان‌هایی به سازنده شی **Employee** ایجاد شده در خط 13 ارسال می‌کند. خط 16 داده شی **Employee** را در خروجی قرار می‌دهد. زمانیکه هر شی **Date** در خطوط 12-11 ایجاد می‌شود، سازنده **Date** تعریف شده در خطوط 11-28 از شکل ۱۱-۱۰ در یک خط خروجی نمایش می‌دهد که سازنده فراخوانی شده است (به سطر اول خروجی نگاه کنید).

```
1  // Fig. 10.11: Date.cpp
2  // Member-function definitions for class Date.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Date.h" // include Date class definition
8
9  // constructor confirms proper value for month; calls
10 // utility function checkDay to confirm proper value for day
11 Date::Date( int mn, int dy, int yr )
12 {
13    if ( mn > 0 && mn <= 12 ) // validate the month
14       month = mn;
15    else
16    {
17       month = 1; // invalid month set to 1
18       cout << "Invalid month (" << mn << ") set to 1.\n";
19    } // end else
20
21    year = yr; // could validate yr
22    day = checkDay( dy ); // validate the day
23
24    // output Date object to show when its constructor is called
25    cout << "Date object constructor for date ";
26    print();
27    cout << endl;
28 } // end Date constructor
29
30 // print Date object in form month/day/year
31 void Date::print() const
32 {
33    cout << month << '/' << day << '/' << year;
34 } // end function print
35
36 // output Date object to show when its destructor is called
37 Date::~Date()
38 {
39    cout << "Date object destructor for date ";
40    print();
41    cout << endl;
42 } // end ~Date destructor
43
44 // utility function to confirm proper day value based on
45 // month and year; handles leap years, too
46 int Date::checkDay( int testDay ) const
47 {
48    static const int daysPerMonth[ 13 ] =
49       { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
50
51    // determine whether testDay is valid for specified month
52    if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
53       return testDay;
54
55    // February 29 check for leap year
56    if ( month == 2 && testDay == 29 && ( year % 400 == 0 ||
57       ( year % 4 == 0 && year % 100 != 0 ) ) )
58       return testDay;
59
60    cout << "Invalid day (" << testDay << ") set to 1.\n";
61    return 1; // leave object in consistent state if bad value
62 } // end function checkDay
```

شکل ۱۱-۱۰ | تعریف تابع عضو کلاس Date.

```
1  // Fig. 10.12: Employee.h
2  // Employee class definition.
3  // Member functions defined in Employee.cpp.
4  #ifndef EMPLOYEE_H
5  #define EMPLOYEE_H
6
```

```
7  #include "Date.h" // include Date class definition
8
9  class Employee
10 {
11 public:
12    Employee( const char * const, const char * const,
13       const Date &, const Date & );
14    void print() const;
15    ~Employee(); // provided to confirm destruction order
16 private:
17    char firstName[ 25 ];
18    char lastName[ 25 ];
19    const Date birthDate; // composition: member object
20    const Date hireDate; // composition: member object
21 }; // end class Employee
22
23 #endif
```

شکل ۱۲-۱۰ | تعریف کلاس Employee که ترکیب را نمایش می‌دهد.

```
1  // Fig. 10.13: Employee.cpp
2  // Member-function definitions for class Employee.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <cstring> // strlen and strncpy prototypes
8  using std::strlen;
9  using std::strncpy;
10
11 #include "Employee.h" // Employee class definition
12 #include "Date.h" // Date class definition
13
14 // constructor uses member initializer list to pass initializer
15 // values to constructors of member objects birthDate and hireDate
16 // [Note: This invokes the so-called "default copy constructor" which the
17 // C++ compiler provides implicitly.]
18 Employee::Employee( const char * const first, const char * const last,
19    const Date &dateOfBirth, const Date &dateOfHire )
20    : birthDate( dateOfBirth ), // initialize birthDate
21      hireDate( dateOfHire ) // initialize hireDate
22 {
23    // copy first into firstName and be sure that it fits
24    int length = strlen( first );
25    length = ( length < 25 ? length : 24 );
26    strncpy( firstName, first, length );
27    firstName[ length ] = '\0';
28
29    // copy last into lastName and be sure that it fits
30    length = strlen( last );
31    length = ( length < 25 ? length : 24 );
32    strncpy( lastName, last, length );
33    lastName[ length ] = '\0';
34
35    // output Employee object to show when constructor is called
36    cout << "Employee object constructor: "
37       << firstName << ' ' << lastName << endl;
38 } // end Employee constructor
39
40 // print Employee object
41 void Employee::print() const
42 {
43    cout << lastName << ", " << firstName << "  Hired: ";
44    hireDate.print();
45    cout << "  Birthday: ";
46    birthDate.print();
47    cout << endl;
48 } // end function print
49
50 // output Employee object to show when its destructor is called
```

```
51 Employee::~Employee()
52 {
53    cout << "Employee object destructor: "
54       << lastName << ", " << firstName << endl;
55 } // end ~Employee destructor
```

شکل ۱۳-۱۰ | تعریف تابع عضو کلاس Employee شامل سازنده با یک لیست مقداردهی کننده عضو.

```
1  // Fig. 10.14: fig10_14.cpp
2  // Demonstrating composition--an object with member objects.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Employee.h" // Employee class definition
8
9  int main()
10 {
11    Date birth( 7, 24, 1949 );
12    Date hire( 3, 12, 1988 );
13    Employee manager( "Bob", "Blue", birth, hire );
14
15    cout << endl;
16    manager.print();
17
18    cout << "\nTest Date constructor with invalid values:\n";
19    Date lastDayOff( 14, 35, 1994 ); // invalid month and day
20    cout << endl;
21    return 0;
22 } // end main
```

```
Date object constructor date 7/24/1949
Date object constructor date 3/12/1988
Employee object constructor: Bob Blue

Blue, Bob  Hired: 3/12/1988 Birthday: 7/24/1949


Test Date constructor with invalid values:
Invalid month (14) set to 1.
Invalid day (35) set to 1.
Date object constructor for date 1/1/1994

Date object destructor for date 1/1/1994
Employee object destructor: Blue, Bob
Date object destructor for date 3/12/1988
Date object destructor for date 7/24/1949
Date object destructor for date 3/12/1988
Date object destructor for date 7/24/1949
```

شکل ۱۴-۱۰ | مقداردهی کننده‌های عضو شی.

کلاس **Date** و کلاس **Employee** هر یک شامل یک نابودکننده هستند (به‌ترتیب خطوط 42-37 از شکل ۱۱-۱۰ و خط 55-51 از شکل ۱۳-۱۰) که به هنگام نابودی یک شی از کلاس مربوطه، یک پیغام چاپ می‌کنند. این امکان به ما اجازه می‌دهد تا توسط خروجی برنامه تایید کنیم که شی‌ها از داخل به خارج ایجاد شده و به ترتیب معکوس از خارج به داخل نابود می‌شوند (یعنی، اعضای عضو **Date** پس از اینکه شی **Employee** که حاوی آنها است، نابود شد از بین می‌روند). در خروجی شکل ۱۴-۱۰ به چهار خط پایانی توجه کنید. دو خط پایانی خروجی اجرای نابود کننده **Date** بر روی شی‌های **hire** (خط 12) و **birth** (خط 11) است. این خروجی‌ها تایید می‌کنند که سه شی ایجاد شده در **main** به‌ترتیب معکوس از

ایجاد شدن، نابود شده‌اند. خطوط چهارم و پنجم از خروجی، نمایشی از اجرای نابودکننده‌ها بر روی شی‌های عضو کلاس **Employee** بنام‌های **hireDate** (شکل ۱۲-۱۰، خط 20) و **birthDate** (شکل ۱۲-۱۰، خط 19) است. این خروجی‌ها تایید می‌کنند که شی **Employee** از خارج به درون نابود می‌شود، یعنی ابتدا نابودکننده **Employee** اجرا می‌شود، سپس شی‌های عضو به ترتیب معکوس از حالتی که ایجاد شده‌اند نابود می‌گردند. مجدداً در خروجی شکل ۱۴-۱۰ خبری از سازنده‌ها برای این شی‌ها نیست، چرا که برای آنها سازنده‌های کپی کننده پیش‌فرض توسط کامپایلر C++ تدارک دیده شده است.

یک شی عضو نیازی به مقداردهی صریح اولیه از طریق یک مقداردهی کننده عضو ندارد. اگر یک مقداردهی کننده عضو در نظر گرفته نشده باشد، بطور ضمنی سازنده پیش‌فرض برای آن شی عضو فراخوانی خواهد شد. مقادیر تدارک دیده شده توسط سازنده پیش‌فرض هر چه باشند، می‌توانند توسط توابع *set* بازنویسی شوند. با این همه، برای مقداردهی‌های پیچیده، چنین روشی مستلزم کار و زمان بیشتری است.

در شکل ۱۱-۱۰ و خط 26، به فراخوانی تابع عضو **print** از **Date** توجه کنید. برخی از توابع عضو در C++ نیازی به آرگومان ندارند. به این دلیل که هر تابع عضو حاوی یک دستگیره (هندل) ضمنی (بفرم یک اشاره‌گر) به شی است که بر روی آن عمل می‌کند. در بخش ۵-۱۰ به معرفی اشاره‌گرهای ضمنی خواهیم پرداخت که توسط کلمه کلیدی **this** معرفی می‌شوند.

کلاس **Employee** از دو آرایه 25 کاراکتری (شکل ۱۲-۱۰، خطوط 18-17) برای عرضه نام و نام خانوادگی کارمند سود می‌برد. امکان دارد این آرایه به هنگام مواجه شدن با اسامی کمتر از 24 کاراکتر، فضای حافظه را تلف کند. همچنین اسامی طولانی‌تر از 24 کاراکتر برای اینکه با سایز آرایه هماهنگ شوند، کوتاه خواهند شد. در بخش ۷-۱۰ به معرفی نسخه دیگری از کلاس **Employee** خواهیم پرداخت که بصورت دینامیکی و دقیقاً به میزان مورد نیاز برای نگهداری نام و نام خانوادگی فضا ایجاد می‌کند. یکی از روش‌های ساده برای عرضه نام و نام خانوادگی یک کارمند استفاده از دو شی رشته (***string***) است که فضای مورد نیاز را تدارک می‌بینند. اگر چنین کاری انجام دهیم، سازنده **Employee** بصورت زیر خواهد بود

```
Employee::Employee(const string &first, const string &last,
    const Date &dateOfBirth, const Date &dateOfHire)
    :firstName(first),//initialize firstName
     lastName(last),// initialize lastName
     birthDate(dateOfBirth),// initialize birthDate
     hireDate(dateOfHire)// initialize hireDate
{
     // output Employee object to show when constructor is called
     cout << "Employee object constructor:"
          <<firstName<<' '<<lastName<<endl;
} // end Employee constructor
```

دقت کنید که اعضای داده **firstName** و **lastName** (شی‌های رشته) از طریق مقداردهی کننده‌های عضو، مقداردهی شده‌اند. کلاس‌های **Employee** معرفی شده در فصل‌های ۱۲ و ۱۳ از شی‌های **string** به این روش استفاده می‌کنند. در این فصل، از رشته‌های مبتنی بر اشاره‌گر استفاده کرده‌ایم تا خواننده با کاربرد اشاره‌گرها بیشتر آشنا شود.

۱۰-٤ توابع و کلاس‌های friend

با اینکه تابع **friend** یک کلاس، خارج از قلمرو کلاس تعریف می‌شود، اما هنوز هم دارای مجوز دسترسی اعضای غیرسراسری (و سراسری) کلاس می‌باشد. توابع منفرد یا کل کلاس‌ها می‌توانند برای کلاس دیگری بصورت **friend** (*دوست*) اعلان شوند.

توابع **friend** می‌توانند در افزایشی کارایی موثر باشند. در این بخش به معرفی یک مثال غیرکاربردی از نحوه عملکرد و توابع **friend** می‌پردازیم. سپس در ادامه این کتاب، از توابع **friend** در عملگرهای سربارگذاری شده برای استفاده با شی‌های کلاس (فصل ۱۱) و ایجاد کلاس‌های تکرار شونده استفاده خواهیم کرد.

برای اعلان یک تابع بعنوان *دوست یک کلاس*، قبل از نمونه اولیه تابع در تعریف کلاس از کلمه کلیدی ***friend*** استفاده می‌شود. برای اعلان تمام توابع عضو کلاس **ClassTwo** بصورت دوستان کلاس **ClassOne**، از اعلان زیر در تعریف کلاس **ClassOne** استفاده می‌شود.

```
friend class ClassTwo
```

دقت کنید که دوستی اهدا می‌شود، اما الزاما پذیرفته نمی‌شود، یعنی کلاس **B** می‌تواند دوست کلاس **A** باشد، اما کلاس **A** بایستی بصورت صریح اعلان کند که کلاس **B** دوست او است. همچنین رابطه دوستی حالت متقارن یا انتقالی ندارد، یعنی اگر کلاس **A** دوست کلاس **B** باشد، و کلاس **B** دوست کلاس **C** باشد، نمی‌توانید استنتاج کنید که کلاس **B** دوست کلاس **A** است (دوستی حالت متقارن ندارد)، و کلاس **C** دوست کلاس **B** است (چرا که دوستی حالت متقارن ندارد) یا اینکه کلاس **A** دوست کلاس **C** است (دوستی حالت انتقالی ندارد).

تغییر در داده private یک کلاس توسط تابع friend

در برنامه شکل ۱۵-۱۰ یک مثال غیرکابردی عرضه شده که در آن تابع دوست setX برای تنظیم داده خصوصی (private) عضو داده x از کلاس **Count** تعریف شده است. دقت کنید که اعلان **friend** (خط 10) در ابتدای تعریف کلاس آورده شده است (بطور قراردادی) حتی قبل از اعلان توابع عضو سراسری (public). توجه کنید که این اعلان **friend** می‌تواند در هر کجای کلاس آورده شود.

```
1  // Fig. 10.15: fig10_15.cpp
2  // Friends can access private members of a class.
3  #include <iostream>
4  using std::cout;
```

```
5  using std::endl;
6
7  // Count class definition
8  class Count
9  {
10    friend void setX( Count &, int ); // friend declaration
11 public:
12    // constructor
13    Count()
14       : x( 0 ) // initialize x to 0
15    {
16       // empty body
17    } // end constructor Count
18
19    // output x
20    void print() const
21    {
22       cout << x << endl;
23    } // end function print
24 private:
25    int x; // data member
26 }; // end class Count
27
28 // function setX can modify private data of Count
29 // because setX is declared as a friend of Count (line 10)
30 void setX( Count &c, int val )
31 {
32    c.x = val; // allowed because setX is a friend of Count
33 } // end function setX
34
35 int main()
36 {
37    Count counter; // create Count object
38
39    cout << "counter.x after instantiation: ";
40    counter.print();
41
42    setX( counter, 8 ); // set x using a friend function
43    cout << "counter.x after call to setX friend function: ";
44    counter.print();
45    return 0;
46 } // end main
```

```
counter.x after instantion: 0
counter.x after call to setX friend function: 8
```

شکل ۱۵-۱۰ | دوستان می‌توانند به اعضای خصوصی یک کلاس دسترسی داشته باشند.

تابع **setX** (خطوط 33-30) یک تابع منفرد به سبک C است و تابع عضوی از کلاس **Count** نمی‌باشد. به همین دلیل، زمانیکه **setX** برای شی **counter** فراخوانی می‌شود، خط 42 مبادرت به ارسال **counter** بعنوان یک آرگومان به **setX** بجای استفاده از یک دستگیره (مانند نام یک شی) برای فراخوانی تابع می‌کند، همانند

```
counter.setX(8);
```

همانطوری که قبلاً هم گفته شد برنامه ۱۵-۱۰ یک برنامه غیرکاربردی است که در آن از **friend** استفاده شده است. مقتضی است که تابع **setX** بصورت یک تابع عضو از کلاس **Count** تعریف شود. همچنین متمایز کردن برنامه ۱۵-۱۰ به سه فایل هم می‌تواند مناسب باشد:

۱ـ فایل سرآیند (مانند *Count.h*) حاوی تعریف کلاس **Count**، که حاوی نمونه اولیه تابع دوست **setX** است.

۲ـ پیاده سازی فایل (مانند *Count.cpp*) حاوی تعاریف توابع عضو کلاس **Count** و تعریف تابع دوست **setX**.

۳- برنامه تست کننده (مانند *fig10_15.cpp*) با **main**.

اشتباه سهوی در تغییر یک عضو خصوصی با یک تابع غیردوست

برنامه شکل ۱۶-۱۰ به بررسی پیغام‌های خطا می‌پردازد که توسط کامپایلر و در زمانیکه تابع غیردوست cannotSetX برای تغییر در داده عضو خصوصی x فراخوانی می‌شود (خطوط 32-29). امکان تصریح توابع سربارگذاری شده به عنوان دوستان کلاس وجود دارد. هر تابع سربارگذاری شده که قصد دارد حالت دوست داشته باشد بایستی بصورت صریح در تعریف کلاس بعنوان دوست کلاسی اعلان شده باشد.

```
1  // Fig. 10.16: fig10_16.cpp
2  // Non-friend/non-member functions cannot access private data of a class.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  // Count class definition (note that there is no friendship declaration)
8  class Count
9  {
10 public:
11    // constructor
12    Count()
13       : x( 0 ) // initialize x to 0
14    {
15       // empty body
16    } // end constructor Count
17
18    // output x
19    void print() const
20    {
21       cout << x << endl;
22    } // end function print
23 private:
24    int x; // data member
25 }; // end class Count
26
27 // function cannotSetX tries to modify private data of Count,
28 // but cannot because the function is not a friend of Count
29 void cannotSetX( Count &c, int val )
30 {
31    c.x = val; // ERROR: cannot access private member in Count
32 } // end function cannotSetX
33
34 int main()
35 {
36    Count counter; // create Count object
37
38    cannotSetX( counter, 3 ); // cannotSetX is not a friend
39    return 0;
40 } // end main
```

Borland C++ command-line compiler error messages

```
Error E2247 Fig10_16/fig10_16.ccp 31:  'Conut::x'  is not accessible in
    function cannotSetX(Count &,int)
```

Microsoft Visual C++.NET compiler error message

```
C:\cpphtp5_examples\ch10\Fig10_16\fig10_16.cpp(31):error C2248: 'Count::x'
   : cannot access private member declared in class 'Count'
        C:\ccphtp5_examples\ch10\Fig10_16\fig10_16.ccp(24):see declaration
           of 'Count::x'
        C:\cpphtp5_examples\ch10\Fig10_16\fig10_16.cpp(9) :see declaration
```

GNU C++ compiler error message

```
fig10_16.ccp:24: error: 'int Count::x' is private
fig10_16.ccp:31: error: within this context
```

شکل ١٦-١٠ | توابع غیردوست/غیرعضو قادر به دسترسی به اعضای خصوصی نمی‌باشند.

۱۰-۵ استفاده از اشاره‌گر this

مشاهده کردید که یک شی از توابع عضو می‌تواند در داده شی دستکاری کند. چگونه توابع عضو می‌دانند که کدام یک از اعضای داده شی را دستکاری کنند؟ هر شی از طریق یک اشاره‌گر بنام **this** (یک کلمه کلیدی در C++) به آدرس متعلق بخود دسترسی دارد. اشاره‌گر **this** یک شی، بخشی از خود شی نمی‌باشد، یعنی سایز حافظه اشغال شده توسط اشاره‌گر **this** تاثیری در نتیجه اجرای **sizeOf** بر روی شی ندارد. بجای آن اشاره‌گر **this** بصورت یک آرگومان ضمنی به هر تابع عضو غیراستاتیک شی ارسال می‌شود (توسط کامپایلر). در بخش ۷-۱۰ به معرفی اعضای کلاس استاتیک و توضیح اینکه چرا اشاره‌گرهای **this** بصورت غیرصریح به توابع عضو استاتیک ارسال می‌شوند، پرداخته شده است.

شی‌ها از اشاره‌گر **this** بصورت ضمنی (که در این بخش آنرا انجام می‌دهیم) یا صریح برای مراجعه اعضای داده و توابع عضو خود استفاده می‌کنند. نوع اشاره‌گر **this** بستگی به نوع شی دارد و خواه تابع عضو که در آن از **this** استفاده شده ثابت باشد یا خیر. برای مثال، در یک تابع عضو غیرثابت از کلاس **Employee**، اشاره‌گر **this** دارای نوع **Employee *const** است (یک ثابت اشاره‌گر به یک شی غیرثابت **Employee**). در تابع عضو ثابت از کلاس **Employee**، اشاره‌گر **this** دارای نوع داده **const Employee * const** است (یک ثابت اشاره‌گر به یک شی ثابت **Employee**). اولین مثال ما در این بخش نمایش استفاده ضمنی و صریح از اشاره‌گر **this** است.

استفاده ضمنی و صریح از اشاره‌گر this برای دسترسی به اعضا داده یک شی

برنامه شکل ۱۷-۱۰ به بیان نحوه استفاده و صریح از اشاره‌گر **this** بر روی یک تابع عضو از کلاس **Test** برای چاپ داده خصوصی **x** از شی **Test** پرداخته است. برای بیان این هدف، ابتدا تابع عضو **print** (خطوط 37-25) مقدار **x** را با استفاده از اشاره‌گر **this** بصورت ضمنی چاپ می‌کند (خط 28)، فقط نام عضو داده مشخص شده است. پس از **print** به دو روش برای دسترسی به x از طریق اشاره‌گر **this** استفاده شده است. عملگر فلش (->) و عملگر نقطه (.).

```
1  // Fig. 10.17: fig10_17.cpp
2  // Using the this pointer to refer to object members.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  class Test
8  {
9  public:
10    Test( int = 0 ); // default constructor
11    void print() const;
12 private:
13    int x;
```

```
14 }; // end class Test
15
16 // constructor
17 Test::Test( int value )
18    : x( value ) // initialize x to value
19 {
20    // empty body
21 } // end constructor Test
22
23 // print x using implicit and explicit this pointers;
24 // the parentheses around *this are required
25 void Test::print() const
26 {
27    // implicitly use the this pointer to access the member x
28    cout << "        x = " << x;
29
30    // explicitly use the this pointer and the arrow operator
31    // to access the member x
32    cout << "\n  this->x = " << this->x;
33
34    // explicitly use the dereferenced this pointer and
35    // the dot operator to access the member x
36    cout << "\n(*this).x = " << ( *this ).x << endl;
37 } // end function print
38
39 int main()
40 {
41    Test testObject( 12 ); // instantiate and initialize testObject
42
43    testObject.print();
44    return 0;
45 } // end main
```

```
        x = 12
   this->x = 12
(*this ).x = 12
```

شکل ۱۷-۱۰ | دسترس ضمنی و صریح اشاره‌گر this به اعضای یک شی.

به پرانتزهای قرار گرفته در اطراف **this*** (خط 36) به هنگام استفاده از عملگر انتخاب عضو (.) توجه کنید. وجود پرانتزها ضروری است چرا که عملگر نقطه به نسبت عملگر * از اولویت بالاتری برخوردار است. بدون حضور پرانتزها، عبارت **this.x*** با خطای کامپایل مواجه خواهد شد، چرا که عملگر نقطه نمی‌تواند با اشاره‌گر بکار گرفته شود.

یکی از نکات جالب در استفاده از اشاره‌گر **this** اجتناب از تخصیص یک شی به خودش است. همانطوری که در فصل یازدهم شاهد خواهید بود، تخصیص بخود می‌تواند خطاهای بسیاری جدی در زمانیکه شی حاوی اشاره‌گرها با فضای اخذ شده دینامیکی باشد، بوجود آورد.

استفاده از اشاره‌گر this برای فراخوانی آبشاری تابع

یکی دیگر از کاربردهای اشاره‌گر **this** فراخوانی *آبشاری* توابع عضو است که در آن توابع مضاعف توسط یک عبارت فراخوانی می‌شوند (همانند خط 14 از برنامه شکل ۲۰-۱۰). برنامه شکل‌های ۱۸-۱۰ الی ۲۰-۱۰ تغییر یافته توابع **setTime**، **setHour**، **setMinute** و **setSecound** از کلاس **Time** هستند که هر یک مراجعه‌ای به یک شی **Time** برگشت می‌دهند تا فراخوانی آبشاری تابع امکان‌پذیر باشد. در شکل

۱۹-۱۰ توجه کنید که آخرین عبارت در بدنه هر یک از این توابع عضو **this*** (خطوط 40، 33، 26 و 47) را بفرم نوع برگشتی **Time &** برگشت می‌دهند.

برنامه شکل ۲۰-۱۰ شی **t** از کلاس **Time** را ایجاد کرده (خط ۱۱)، سپس از آن در فراخوانی آبشاری تابع عضو استفاده می‌کند (خطوط 14 و 26). عملگر نقطه (.) از چپ به راست ارزیابی می‌شود، از اینرو ابتدا خط 14 مبادرت به ارزیابی **t.setHour(18)** کرده سپس مراجعه‌ای به شی **t** بعنوان مقدار فراخوانی این تابع برگشت می‌دهد. سپس مابقی عبارت بصورت زیر تفسیر می‌گردد.

```
t.setMinute(30).setSecuond(22);
```

فراخوانی **t.setMinute(30)** اجرا شده و یک مراجعه به شی **t** برگشت می‌دهد. مابقی عبارت بصورت زیر تفسیر می‌شود

```
t.setSecound(22);
```

همچنین خط 26 نیز از آبشاری استفاده می‌کند. فراخوانی باید به ترتیب ظاهر شده در خط 26 انجام شود، چرا که **printStandard** تعریف شده در کلاس مراجعه‌ای به **t** برگشت نمی‌دهد. فراخوانی **printStandard** قبل از فراخوانی **setTime** در خط 26 خطای کامپایل بدنبال خواهد داشت. در فصل یازدهم چندین مثال در ارتباط با فراخوانی آبشاری توابع آورده شده است. در یکی از مثال‌ها از عملگرها >> به‌همراه **cout** استفاده شده تا مقادیر مضاعف در یک عبارت چاپ شوند.

```
1  // Fig. 10.18: Time.h
2  // Cascading member function calls.
3
4  // Time class definition.
5  // Member functions defined in Time.cpp.
6  #ifndef TIME_H
7  #define TIME_H
8
9  class Time
10 {
11 public:
12    Time( int = 0, int = 0, int = 0 ); // default constructor
13
14    // set functions (the Time & return types enable cascading)
15    Time &setTime( int, int, int ); // set hour, minute, second
16    Time &setHour( int ); // set hour
17    Time &setMinute( int ); // set minute
18    Time &setSecond( int ); // set second
19
20    // get functions (normally declared const)
21    int getHour() const; // return hour
22    int getMinute() const; // return minute
23    int getSecond() const; // return second
24
25    // print functions (normally declared const)
26    void printUniversal() const; // print universal time
27    void printStandard() const; // print standard time
28 private:
29    int hour; // 0 - 23 (24-hour clock format)
30    int minute; // 0 - 59
31    int second; // 0 - 59
32 }; // end class Time
33
34 #endif
```

شکل ۱۸-۱۰ | تعریف کلاس Time اصلاح شده تا فراخوانی آبشاری تابع عضو امکان‌پذیر شود.

```
1  // Fig. 10.19: Time.cpp
2  // Member-function definitions for Time class.
3  #include <iostream>
4  using std::cout;
5
6  #include <iomanip>
7  using std::setfill;
8  using std::setw;
9
10 #include "Time.h" // Time class definition
11
12 // constructor function to initialize private data;
13 // calls member function setTime to set variables;
14 // default values are 0 (see class definition)
15 Time::Time( int hr, int min, int sec )
16 {
17    setTime( hr, min, sec );
18 } // end Time constructor
19
20 // set values of hour, minute, and second
21 Time &Time::setTime( int h, int m, int s ) // note Time & return
22 {
23    setHour( h );
24    setMinute( m );
25    setSecond( s );
26    return *this; // enables cascading
27 } // end function setTime
28
29 // set hour value
30 Time &Time::setHour( int h ) // note Time & return
31 {
32    hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
33    return *this; // enables cascading
34 } // end function setHour
35
36 // set minute value
37 Time &Time::setMinute( int m ) // note Time & return
38 {
39    minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
40    return *this; // enables cascading
41 } // end function setMinute
42
43 // set second value
44 Time &Time::setSecond( int s ) // note Time & return
45 {
46    second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
47    return *this; // enables cascading
48 } // end function setSecond
49
50 // get hour value
51 int Time::getHour() const
52 {
53    return hour;
54 } // end function getHour
55
56 // get minute value
57 int Time::getMinute() const
58 {
59    return minute;
60 } // end function getMinute
61
62 // get second value
63 int Time::getSecond() const
64 {
65    return second;
66 } // end function getSecond
67
68 // print Time in universal-time format (HH:MM:SS)
69 void Time::printUniversal() const
70 {
```

```
71    cout << setfill( '0' ) << setw( 2 ) << hour << ":"
72       << setw( 2 ) << minute << ":" << setw( 2 ) << second;
73 } // end function printUniversal
74
75 // print Time in standard-time format (HH:MM:SS AM or PM)
76 void Time::printStandard() const
77 {
78    cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
79       << ":" << setfill( '0' ) << setw( 2 ) << minute
80       << ":" << setw( 2 ) << second << ( hour < 12 ? " AM" : " PM" );
81 } // end function printStandard
```

شکل ۱۹-۱۰ | تعریف تابع عضو کلاس Time اصلاح شده تا فراخوانی آبشاری تابع عضو امکان‌پذیر شود.

```
1  // Fig. 10.20: fig10_20.cpp
2  // Cascading member function calls with the this pointer.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Time.h" // Time class definition
8
9  int main()
10 {
11    Time t; // create Time object
12
13    // cascaded function calls
14    t.setHour( 18 ).setMinute( 30 ).setSecond( 22 );
15
16    // output time in universal and standard formats
17    cout << "Universal time: ";
18    t.printUniversal();
19
20    cout << "\nStandard time: ";
21    t.printStandard();
22
23    cout << "\n\nNew standard time: ";
24
25    // cascaded function calls
26    t.setTime( 20, 20, 20 ).printStandard();
27    cout << endl;
28    return 0;
29 } // end main
```

```
Universal time: 18:30:22
Standard time: 6:30:22 PM

New standard time: 8:20:20 PM
```

شکل ۲۰-۱۰ | فراخوانی آبشاری تابع عضو.

۱۰-۶ مدیریت دینامیکی حافظه با عملگرهای new و delete

زبان C++ به برنامه‌نویسان امکان داده تا بر نحوه اخذ و ترخیص حافظه در برنامه‌ها برای هر نوع داده توکار (built-in) یا تعریف شده توسط کاربر کنترل داشته باشند. به اینحالت *مدیریت دینامیکی حافظه* گفته می‌شود و توسط عملگرهای **new** و **delete** انجام می‌گردد. بخاطر دارید که کلاس **Employee** (شکل ۱۰-۱۲ و ۱۰-۱۳) از دو آرایه 25 کاراکتری برای عرضه نام و نام خانوادگی کارمند استفاده می‌کرد. تعریف کلاس **Employee** (شکل ۱۰-۱۲) بایستی تعداد عناصر در هر کدامیک از این آرایه‌ها را در زمان اعلان آنها بعنوان داده‌های عضو مشخص می‌کرد، چرا که سایز عضوهای داده میزان حافظه مورد نیاز برای ذخیره یک شی **Employee** را دیکته می‌کرد. همانطوری‌که گفتیم، این آرایه‌ها می‌توانند در برخورد با

اسامی کوچکتر از 24 کاراکتر، فضای حافظه را تلف کنند. همچنین اسامی بزرگتر از 24 کاراکتر بایستی به منظور قالب شدن در این آرایه‌ها با سایز مشخص شده، قطع کردند.

بهتر نیست از آرایه‌های استفاده کنیم که دقیقاً به تعداد عناصر مورد نیاز مبادرت به ذخیره نام و نام‌خانوادگی کارمندی می‌کنند؟ مدیریت دینامیکی حافظه امکان می‌دهد تا دقیقاً همین کار را انجام دهیم. همانطوری‌که در مثال بخش ۷-۱۰ خواهید دید، اگر اعضای داده آرایه **firstName** و **lastName** را با اشاره‌گرهای به **char** جایگزین سازیم، می‌توانیم از عملگر **new** برای اخذ دینامیکی (رزرو) حافظه به میزان دقیق و مورد نیاز برای نگهداری هر نام در زمان اجرا استفاده کنیم. مدیریت دینامیکی حافظه به این روش سبب ایجاد آرایه در فضای آزاد ذخیره‌سازی (غالباً *heap* نامیده می‌شود) می‌شود، ناحیه‌ای از حافظه تخصیص یافته به هر برنامه به منظور ذخیره‌سازی شی‌های ایجاد شده در زمان اجرا. زمانیکه حافظه برای یک آرایه در heap اخذ شد، می‌توانیم با اشاره دادن یک اشاره‌گر به اولین عنصر آرایه، به آن دست پیدا کنیم. زمانیکه دیگر نیازی به آرایه نداریم، می‌توانیم با استفاده از عملگر **delete** حافظه اخذ شده را آزاد کرده و به heap بازگردانیم. در صورت نیاز به حافظه می‌توانیم توسط عملگر **new** دوباره به آن دست پیدا کنیم.

مجدداً به سراغ کلاس **Employee** می‌رویم که به بررسی آن در بخش ۷-۱۰ خواهیم پرداخت. ابتدا، به بررسی جزئیات استفاده از عملگرهای **new** و **delete** در اخذ دینامیکی حافظه می‌پردازیم تا شی‌ها، نوع‌های بنیادین و آرایه‌ها را در آن مکان ذخیره سازیم.

به اعلان و عبارت زیر توجه کنید:

```
Time *timePtr;
timePtr = new Time;
```

عملگر **new** فضای با سایز مناسب برای یک شی از نوع **Time** اخذ می‌کند، سازنده پیش‌فرض برای مقداردهی اولیه شی فراخوانی شده و یک اشاره‌گر از نوع مشخص شده برگشت می‌یابد (یعنی یک ***Time**). توجه کنید که **new** می‌تواند برای اخذ دینامیکی هر نوع داده بنیادین (همانند **int** یا **double**) یا نوع کلاس بکار گرفته شود. اگر **new** قادر به یافتن فضای کافی در حافظه برای شی نباشد، با به راه انداختن یک استثناء نشان می‌دهد که خطائی رخ داده است. در فصل شانزدهم به بررسی استثناءها و رسیدگی به مشکلات رخ داده با **new** خواهیم پرداخت. در عمل نشان خواهیم داد که چگونه می‌توان یک استثناء رخ داده را گرفتار کرد. زمانیکه برنامه‌ای نتواند یک استثناء رخ داده را گرفتار سازد، بلافاصله خاتمه می‌یابد. برای حذف (آزاد کردن) حافظه اخذ شده توسط یک شی، از عملگر **delete** بصورت زیر استفاده می‌کنیم:

```
delete timePtr;
```

این عبارت ابتدا نابودکننده را بر روی شیی که **timePtr** به آن اشاره می‌کند فراخوانی کرده، سپس حافظه اخذ شده توسط آن شی را باز می‌گرداند. پس از اجرای این عبارت، حافظه برگشتی می‌تواند توسط سیستم در اختیار سایر شی‌ها قرار داده شود.

خطای برنامه‌نویسی

عدم رهاسازی حافظه اخذ شده دینامیکی در زمانیکه دیگر به آن نیازی نیست، می‌تواند سیستم را با مشکل «فقدان حافظه» مواجه سازد.

زبان C++ امکان تدارک دیدن یک مقداردهی کننده برای متغیرهای از نوع بنیادین جدیداً ایجاد شده می‌دهد، همانند

```
double *ptr = new double(3.14159);
```

در عبارت فوق، **double** ایجاد شده با 3.14159 مقداردهی اولیه شده و نتیجه به اشاره‌گر **ptr** تخصیص می‌یابد. از همین گرامر می‌توان در لیستی از آرگومان‌ها با کاماهای متمایز شده از یکدیگر در سازنده یک شی استفاده کرد. برای مثال،

```
Time *timePtr = new Time(12,45,0);
```

مبادرت به مقداردهی اولیه شی جدید **Time** با 12:45PM کرده و نتیجه به اشاره‌گر **timePtr** تخصیص می‌یابد.

همانطوری‌که قبلاً هم گفته شد، عملگر **new** می‌تواند در اخذ آرایه‌های دینامیکی بکارگرفته شود. برای مثال، یک آرایه 10 عضوی از نوع صحیح می‌تواند با عبارت زیر اخذ شده و به **gradeArray** تخصیص یابد:

```
int *gradesArray = new int[10];
```

اشاره‌گر **gradesArray** اعلان شده و آن به اشاره‌گری که به اولین عنصر از آرایه 10 عنصری از نوع صحیح که بصورت دینامیکی اخذ شده تخصیص داده می‌شود. بخاطر دارید که سایز یک آرایه باید در زمان کامپایل و با استفاده از یک ثابت صحیح مشخص شده باشد. با این وجود، سایز آرایه اخذ شده دینامیکی می‌تواند با استفاده از هر عبارت صحیح که می‌تواند در زمان اجرا ارزیابی گردد، تعیین شود. همچنین به این نکته توجه داشته باشید که به هنگام اخذ یک آرایه با شی‌های دینامیکی، برنامه نویس نمی‌تواند آرگومان‌هائی به هر سازنده شی ارسال کند. بجای آن، هر شی در آرایه با سازنده پیش‌فرض خود مقداردهی اولیه می‌شود. برای حذف آرایه اخذ شده دینامیکی که **gradesArray** به آن اشاره می‌کند، از عبارت زیر استفاده می‌کنیم.

```
delete[] gradesArray;
```

عبارت فوق حافظه اخذ شده توسط آرایه‌ای که **gradesArray** به آن اشاره می‌کند را آزاد می‌سازد. اگر اشاره‌گر فوق به آرایه‌ای از شی‌ها اشاره داشته باشد، ابتدا نابودکننده برای هر شی موجود در آرایه

فراخوانی می‌شود، سپس حافظه رها می‌گردد. اگر عبارت فوق فاقد براکت‌ها (**[]**) باشد و **gradesArray** به آرایه‌ای از شی‌ها اشاره داشته باشد، فقط اولین شی در آرایه انتخاب و نابود می‌شود.

خطای برنامه‌نویسی

*استفاده از **delete** بجای **[]delete** در ارتباط با آرایه‌ی از شی‌ها می‌تواند خطاهای زمان اجرا بدنبال داشته باشد.*

۱۰-۷ اعضای static کلاس

یک استثناء مهم در قانونی وجود دارد که می‌گوید هر شی از کلاس دارای یک کپی از تمام اعضای داده خود در کلاس است. در موارد خاصی، فقط یک کپی از یک متغیر باید توسط تمام شی‌های کلاس به اشتراک گذاشته شود. از عضو داده استاتیک به همین منظور و دلایل دیگر استفاده می‌شود. چنین متغیری نشاندهنده اطلاعات «در سطح کلاس» است (یعنی خصیصه‌ای از کلاس که مابین تمام نمونه‌ها به اشتراک گذاشته می‌شود، و نه خصیصه یک شی خاص از کلاس). اعلان یک عضو استاتیک با کلمه کلیدی **static** آغاز می‌شود. از نسخه‌های کلاس **GradeBook** در فصل هفتم بخاطر دارید که از اعضای داده استاتیک برای ذخیره‌سازی ثابت‌های نشاندهنده تعداد امتیازات (نمرات) که کلیه شی‌های **GradeBook** می‌توانند نگهداری کنند، استفاده کردیم.

اجازه دهید بحث را با مثالی که در ارتباط با داده استاتیکی و در سطح کلاس است، ادامه دهیم. فرض کنید که یک بازی ویدئوی با موضوع نبرد مریخی‌ها و دیگر مخلوقات فضایی داریم. هر مریخی مایل است تا شجاع بوده و راغب به حمله‌ور شدن به دیگر مخلوقات فضایی در مواقعی است که بداند در صحنه حداقل پنج مریخی دیگر حضور دارند. اگر کمتر از پنج مریخی در صحنه حضور داشته باشند، هر مریخی تبدیل به یک ترسو می‌شود.

از اینرو هر مریخی نیاز دارد تا از تعداد مریخی‌ها (**martianCount**) مطلع باشد. می‌توانیم به هر نمونه از کلاس **Martian** یک **martianCount** بعنوان یک عضو داده اعطا کنیم. اگر چنین کاری انجام دهیم، هر مریخی دارای یک کپی متمایز از عضو داده خواهد بود. هر زمان که یک مریخی جدید ایجاد کنیم، مجبور هستیم تا عضو داده **martianCount** در تمام شی‌های **Martian** را به روز کنیم. انجام اینکار مستلزم این است که هر شی **Martian** دارای یا دسترسی به، دستگیرهای به تمام دیگر شی‌های **Martian** در حافظه داشته باشد. انجام چنین کاری به معنی اتلاف حافظه با کپی‌های که افزونگی ایجاد می‌کنند بوده و زمان هم در این بین تلف می‌شود. بجای اینکار، **martianCount** را بصورت **static** اعلان می‌کنیم. چنین حالتی **martianCount** را به داده‌ای در سطح کلاس تبدیل می‌کند. هر مریخی می‌تواند در صورتیکه عضوی از **Martian** باشد، به **martianCount** دسترسی پیدا کند، و این در صورتی است که

فقط یک کپی از متغیر استاتیکی **martianCount** توسط C++ نگهداری می‌شود. با اینکار در فضای حافظه صرفه‌جویی می‌شود. با افزایش مقدار متغیر استاتیکی **martianCount** توسط سازنده **Martian** و کاستن از مقدار **martianCount** توسط نابودکننده **Martian** در زمان هم صرفه‌جویی می‌کنیم. به‌دلیل وجود یک کپی، مجبور نیستیم تا کپی‌های مجزا از **martianCount** را برای هر شی **Martian** افزایش یا کاهش دهیم.

اگر چه ممکن است اینحالت شبیه متغیرهای سراسری بنظر برسد، اما اعضای داده استاتیکی یک کلاس دارای *قلمرو کلاس* هستند. همچنین اعضای استاتیک می‌توانند بصورت **public**، **private** و **protected** اعلان شوند. یک عضو داده استاتیکی از نوع بنیادین بطور پیش‌فرض با صفر مقداردهی اولیه می‌شود. اگر بخواهید آنرا با مقدار دیگر مقداردهی کنید، عضو داده استاتیکی فقط یکبار مقداردهی اولیه خواهد شد. یک عضو داده استاتیکی ثابت از نوع **int** یا **enum** می‌تواند در اعلان خود در تعریف کلاس مقداردهی اولیه شود. با این همه، دیگر اعضای داده استاتیکی بایستی در قلمرو فایل تعریف شوند (خارج از بدنه تعریف کلاس) و فقط می‌تواند در تعریف آنها مقداردهی اولیه گردند. دقت کنید که اعضای داده استاتیک از نوع کلاس (شی‌های عضو استاتیک) که دارای سازنده‌های پیش‌فرض هستند نیازی به مقداردهی اولیه ندارند چرا که سازنده‌های پیش فرض برای آنها فراخوانی خواهند شد. اعضای استاتیک **private** و **protected** معمولاً از طریق توابع عضو **public** کلاس یا از طریق **friend** (دوستان) کلاس در دسترس قرار می‌گیرند. (در فصل دوازدهم، خواهید دید که اعضای استاتیکی **private** و **protected** می‌توانند از طریق توابع **protected** (محافظت شده) هم در دسترس قرار گیرند). اعضای استاتیکی یک کلاس حتی در زمانیکه شی‌های که از آن کلاس وجود ندارند، وجود دارند. برای دسترسی به یک عضو کلاس استاتیکی **public** در زمانیکه هیچ شی از آن کلاس وجود ندارد، کافیست پیشوند نام کلاس و عملگر باینری تفکیک قلمرو (::) در کنار نام عضو داده آورده شود. برای مثال، اگر متغیر **martianCount** سراسری (public) باشد، می‌توان از طریق عبارت **Martian::martianCount** در زمانیکه هیچ شی از **Martian** وجود ندارد، به آن دسترسی پیدا کرد.

همچنین می‌توان به اعضای کلاس استاتیکی **public** از طریق هر شی از آن کلاس با استفاده از نام شی، عملگر نقطه و نام عضو دسترسی پیدا کرد (مثلاً **MyMartian.martianCount**). برای دسترسی به یک عضو کلاس استاتیک **private** یا **protected** در زمانیکه شی وجود ندارد، یک تابع عضو استاتیک تدارک دیده و تابع با پیشوند نام خود به همراه نام کلاس و عملگر تفکیک قلمرو فراخوانی می‌شود. یک تابع عضو استاتیک سرویسی برای کلاس می‌باشد و نه یک شی خاص از آن کلاس.

برنامه شکل‌های ۲۱-۱۰ الی ۲۳-۱۰ به بیان یک داده عضو استاتیکی خصوصی بنام **count** (شکل ۲۱-۱۰، خط 21) و یک تابع عضو استاتیک سراسری بنام **getCount** (شکل ۲۱-۱۰، خط 15) می‌پردازد. در شکل ۲۲-۱۰، خط 14 مبادرت به تعریف و مقداردهی اولیه داده عضو **count** با صفر در قلمرو فایل کرده و خطوط 21-18 تابع عضو استاتیک **getCount** را تعریف کرده‌اند. توجه کنید که خواه خط 14 یا خط 18 حاوی کلمه کلیدی **static** باشند یا نباشند، هنوز هم هر دو خط به اعضای کلاس استاتیک اشاره دارند. زمانیکه **static** بر روی یک ایتم در قلمرو فایل اعمال می‌شود، آن ایتم فقط در آن فایل شناخته خواهد شد. نیاز است تا اعضای استاتیک یک کلاس از طریق کد هر سرویس‌گیرنده‌ای که به فایل دسترسی دارند، در اختیار آنها قرار داشته باشند، از اینرو نمی‌توانیم آنها را در فایل **.cpp** بصورت **static** اعلان کنیم، فقط می‌توانیم آنها را در فایل **.h** بصورت **static** اعلان نمائیم. عضو داده **count** شمارنده‌ای از تعداد شی‌های کلاس **Employee** است که نمونه‌سازی شده‌اند. زمانیکه شی‌های از کلاس **Employee** وجود دارند، عضو **count** می‌تواند از طریق هر تابع عضو از یک شی **Employee** مورد مراجعه قرار گیرد. در شکل ۲۲-۱۰، **count** توسط هر دو خط 33 در سازنده و خط 48 در نابودکننده مورد مراجعه قرار می‌گیرد. همچنین توجه کنید از آنجا که **count** یک **int** است، می‌توانست در فایل سرآیند در خط 21 از شکل ۲۱-۱۰ مقداردهی اولیه گردد.

خطای برنامه‌نویسی

*قرار دادن کلمه کلیدی **static** در تعریف اعضای داده استاتیکی در قلمرو فایل، خطای کامپایل است.*

در شکل ۲۲-۱۰ به نحوه استفاده از عملگر **new** (خطوط 27 و 30) در سازنده **Employee** به منظور اخذ دینامیکی حافظه به میزان مورد نیاز برای اعضای **firstName** و **lastName** توجه کنید. اگر عملگر **new** قادر به اخذ فضای مورد تقاضا از حافظه برای یک یا هر دو این آرایه‌ها نباشد، بلافاصله برنامه خاتمه می‌یابد. در فصل شانزدهم مکانیزم بهتری برای مواجه شدن با چنین وضعیت‌های در نظر خواهیم گرفت.

```
1  // Fig. 10.21: Employee.h
2  // Employee class definition.
3  #ifndef EMPLOYEE_H
4  #define EMPLOYEE_H
5
6  class Employee
7  {
8  public:
9     Employee( const char * const, const char * const ); // constructor
10    ~Employee(); // destructor
11    const char *getFirstName() const; // return first name
12    const char *getLastName() const; // return last name
13
14    // static member function
15    static int getCount(); // return number of objects instantiated
16 private:
17    char *firstName;
18    char *lastName;
19
20    // static data
```

```
21    static int count; // number of objects instantiated
22 }; // end class Employee
23
24 #endif
```

شکل ۲۱-۱۰ | تعریف کلاس Employee با عضو داده استاتیک برای ردگیری تعداد شی‌های Employee در حافظه.

در شکل ۲۲-۱۰ به پیاده سازی توابع **getFirstName** (خطوط 58-52) و **getLastName** (خطوط 61-67) که اشاره‌گرهای به داده کاراکتری ثابت (**const**) برگشت می‌دهند، دقت کنید.

```
1  // Fig. 10.22: Employee.cpp
2  // Member-function definitions for class Employee.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <cstring> // strlen and strcpy prototypes
8  using std::strlen;
9  using std::strcpy;
10
11 #include "Employee.h" // Employee class definition
12
13 // define and initialize static data member at file scope
14 int Employee::count = 0;
15
16 // define static member function that returns number of
17 // Employee objects instantiated (declared static in Employee.h)
18 int Employee::getCount()
19 {
20    return count;
21 } // end static function getCount
22
23 // constructor dynamically allocates space for first and last name and
24 // uses strcpy to copy first and last names into the object
25 Employee::Employee( const char * const first, const char * const last )
26 {
27    firstName = new char[ strlen( first ) + 1 ];
28    strcpy( firstName, first );
29
30    lastName = new char[ strlen( last ) + 1 ];
31    strcpy( lastName, last );
32
33    count++; // increment static count of employees
34
35    cout << "Employee constructor for " << firstName
36       << ' ' << lastName << " called." << endl;
37 } // end Employee constructor
38
39 // destructor deallocates dynamically allocated memory
40 Employee::~Employee()
41 {
42    cout << "~Employee() called for " << firstName
43       << ' ' << lastName << endl;
44
45    delete [] firstName; // release memory
46    delete [] lastName; // release memory
47
48    count--; // decrement static count of employees
49 } // end ~Employee destructor
50
51 // return first name of employee
52 const char *Employee::getFirstName() const
53 {
54    // const before return type prevents client from modifying
55    // private data; client should copy returned string before
56    // destructor deletes storage to prevent undefined pointer
57    return firstName;
```

```
58 } // end function getFirstName
59
60 // return last name of employee
61 const char *Employee::getLastName() const
62 {
63    // const before return type prevents client from modifying
64    // private data; client should copy returned string before
65    // destructor deletes storage to prevent undefined pointer
66    return lastName;
67 } // end function getLastName
```

شکل ۲۲-۱۰ | تعریف تابع عضو کلاس Employee.

در این پیاده‌سازی، اگر سرویس‌گیرنده مایل به نگهداری یک کپی از نام و یا نام خانوادگی داشته باشد، سرویس‌گیرنده مسئول کپی کردن حافظه اخذ شده دینامیکی در شی **Employee** پس از بدست آوردن اشاره‌گر به داده کاراکتری ثابت از شی است. همچنین امکان پیاده‌سازی **getFirstName** و **getLastName** وجود دارد، از اینرو سرویس‌گیرنده مستلزم ارسال آرایه کاراکتری و سایز آن به هر تابع است. پس توابع می‌توانند نام یا نام خانوادگی را به آرایه کاراکتری تدارک دیده شده توسط سرویس‌گیرنده کپی کنند. توجه کنید که می‌توانیم از کلاس **string** برای برگشت دادن کپی از یک شی رشته به فراخوان به جای برگشت دادن یک اشاره‌گر به داده خصوصی استفاده کنیم.

در شکل ۳۳-۱۰ از تابع عضو استاتیک **getCount** برای تعیین تعداد شی‌های **Employee** موجود استفاده شده است. توجه کنید زمانیکه هیچ شیء در برنامه ایجاد نشده باشد، فراخوانی تابع **Employee::getCount()** صورت می‌گیرد (خط 14 و 38). با این وجود زمانیکه شی‌هائی ایجاد شده باشند، تابع **getCount** می‌تواند از طریق شی‌ها، همانند عبارت موجود در خطوط 23-22 که از اشاره‌گر **e1Ptr** برای فراخوانی تابع **getCount** استفاده شده، فراخوانی گردد. در خط 23 به نحوه استفاده از **e2Ptr->getCount()** یا **Employee::getCount()** توجه کنید که هر دو نتیجه مشابهی بدست می‌دهند چرا که getCount همیشه به همان عضو استاتیک **count** دسترسی دارد.

```
1  // Fig. 10.23: fig10_23.cpp
2  // Driver to test class Employee.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Employee.h" // Employee class definition
8
9  int main()
10 {
11    // use class name and binary scope resolution operator to
12    // access static number function getCount
13    cout << "Number of employees before instantiation of any objects is "
14       << Employee::getCount() << endl; // use class name
15
16    // use new to dynamically create two new Employees
17    // operator new also calls the object's constructor
18    Employee *e1Ptr = new Employee( "Susan", "Baker" );
19    Employee *e2Ptr = new Employee( "Robert", "Jones" );
20
21    // call getCount on first Employee object
22    cout << "Number of employees after objects are instantiated is "
23       << e1Ptr->getCount();
```

```
24
25    cout << "\n\nEmployee 1: "
26       << e1Ptr->getFirstName() << " " << e1Ptr->getLastName()
27       << "\nEmployee 2: "
28       << e2Ptr->getFirstName() << " " << e2Ptr->getLastName() << "\n\n";
29
30    delete e1Ptr; // deallocate memory
31    e1Ptr = 0; // disconnect pointer from free-store space
32    delete e2Ptr; // deallocate memory
33    e2Ptr = 0; // disconnect pointer from free-store space
34
35    // no objects exist, so call static member function getCount again
36    // using the class name and the binary scope resolution operator
37    cout << "Number of employees after objects are deleted is "
38       << Employee::getCount() << endl;
39    return 0;
40 } // end main
```

```
Number of employees before instantiation of any object is 0
Employee constructor for Susan Baker called.
Employee constructor for Bahram Jones called.
Number of Employee after objects are instantiated is 2

Employee 1: Susan Barker
Employee 2: Robert Jones

~ Employee() called for Susan Baker
~ Employee() called for Robert Jones
Number of Employee after objects are deleted is 0
```

شکل ۲۳-۱۰ |عضو داده استاتیک تعداد شی‌های موجود در کلاس را روگیری می‌کند.

اگر تابع عضو به اعضای داده غیراستاتیک یا توابع عضو غیراستاتیک یک کلاس دسترسی ندارد، باید بصورت **static** اعلان شود. برخلاف توابع عضو غیراستاتیک، یک تابع عضو استاتیک دارای اشاره‌گر **this** نمی‌باشد، چرا که اعضای داده استاتیک و توابع عضو استاتیک بصورت مستقل از هر شی کلاس می‌باشند. بایستی اشاره‌گر **this** به یک شی خاص کلاس اشاره داشته باشد و زمانیکه یک تابع عضو استاتیک فراخوانی می‌شود، امکان وجود هر شی از آن کلاس در حافظه وجود ندارد.

خطای برنامه‌نویسی

*استفاده از اشاره‌گر **this** در یک تابع استاتیک خطای کامپایل بدنبال خواهد داشت.*

در خطوط 19-18 از شکل ۲۳-۱۰ از عملگر **new** برای اخذ دینامیکی حافظه برای دو شی **Employee** استفاده شده است. بخاطر دارید که اگر برنامه قادر به اخذ حافظه برای یک یا هر دو این شی‌ها نشود، بلافاصله پایان می‌یابد. پس از اخذ حافظه برای هر شی **Employee**، سازنده آن فراخوانی می‌شود. زمانیکه از **delete** در خطوط 30 و 32 برای بازپس‌گیری حافظه اخذ شده توسط دو شی **Employee** استفاده می‌شود، نابودکننده این شی‌ها فراخوانی می‌گردند.

۸-۱۰ انتزاع داده و پنهان سازی اطلاعات

۱-۸-۱۰ مثال: نوع داده انتزاعی آرایه

در فصل هفتم به بررسی آرایه‌ها پرداختیم. همانطوری که در آنجا توضیح دادیم، آرایه چیزی بیش از یک اشاره‌گر و مقداری فضا در حافظه نیست. در صورتیکه برنامه‌نویس متوجه باشد می‌تواند از این قابلیت اولیه

به هنگام کار بر روی آرایه استفاده کند. عملیات‌های متعددی می‌توان بر روی آرایه‌ها انجام داده، اما همه آنها بصورت توکار در C++ تعبیه نشده‌اند. با کلاس‌های C++، برنامه‌نویس می‌تواند یک آرایه ADT توسعه دهد که به آرایه‌های اولیه (خام) ترجیح داده می‌شوند. کلاس آرایه می‌تواند قابلیت‌های جدیدی داشته و آنها را در اختیار کاربر خود قرار دهد، همانند:

- بررسی محدودهٔ شاخص
- محدودهٔ اختیاری شاخص بجای شروع شدن از صفر
- تخصیص آرایه
- مقایسه آرایه
- ورودی/خروجی آرایه
- آرایه‌های که از سایز خود مطلع باشند
- آرایه‌های که بصورت دینامیکی خود را با عناصر بیشتر تطبیق می‌دهند
- آرایه‌های که می‌توانند از خود بصورت مرتب در فرمت جدولی چاپ بگیرند.

در فصل یازدهم، چنین کلاس آرایه‌ای را با قابلیت‌های فوق ایجاد خواهیم کرد. بخاطر دارید که کلاس الگوی **vector** از کتابخانه استاندارد C++ (فصل هفتم) برخی از این قابلیت‌ها را به همان اندازه فراهم می‌کرد.

۱۰-۸-۲ مثال: نوع داده انتزاعی رشته

زبان C++ عمداً یک زبان پراکنده است که برای برنامه‌نویسان قابلیت‌های خام در نظر گرفته تا برنامه‌نویسان برحسب نیاز از آنها بعنوان ابزار در ایجاد سیستم‌های عریض و طویل استفاده کنند. زبان برای کاستن از هزینه کارایی طراحی شده است. زبان C++ مناسب برای برنامه‌نویسی کاربردی و برنامه‌نویسی سیستم است. مطمئناً، امکان قراردادن نوع داده رشته در میان انواع داده توکار C++ وجود داشت. بجای اینکار، زبان برای در برگرفتن مکانیزمی برای ایجاد و پیاده‌سازی نوع داده انتزاعی رشته از طریق کلاس‌ها طراحی شده است. در فصل سوم به معرفی کلاس **string** از کتابخانه C++ پرداختیم و در فصل یازدهم مبادرت به ایجاد رشته ADT متعلق بخودمان خواهیم‌کرد. در فصل هیجدهم، کلاس **string** به تفصیل توضیح داده شده است.

۱۰-۹ کلاس‌های حامل و تکرارشونده‌ها

در میان انواع کلاس‌های پرطرفدار، *کلاس‌های حامل* از جایگاه خاصی برخوردارند. به این کلاس‌ها گاهاً *کلاس‌های کلکسیون* هم گفته می‌شود، یعنی کلاس‌های که برای نگهداری کلکسیونی از شی‌ها طراحی شده‌اند. معمولاً کلاس‌های حامل سرویس‌های همانند درج، حذف، جستجو، مرتب‌سازی و تست یک

ایتم برای تعیین اینکه آیا عضوی از کلکسیون است یا خیر، ارائه می‌دهند. آرایه‌ها، پشته‌ها، صف‌ها، درخت‌ها و لیست‌های پیوندی نمونه‌های از کلاس‌های حامل هستند. در فصل هفتم در ارتباط با آرایه مطالبی آموختیم و در فصل ۲۱ به بررسی ساختمان‌های داده دیگر خواهیم پرداخت.

شریک دانستن شی‌های تکرار شونده با کلاس‌های حامل معقول به نظر می‌رسد. یک تکرار شونده، شی است که در میان یک کلکسیون «قدم» می‌زند، و ایتم بعدی را برگشت می‌دهد (یا عملیاتی بر روی ایتم بعدی انجام می‌دهد). زمانیکه یک تکرار شونده برای کلاسی در نظر گرفته می‌شود، بدست آوردن عنصر بعدی از آن کلاس کار آسانی می‌شود. یک کلاس حامل می‌تواند چندین تکرار شونده داشته باشد. هر تکرار شونده مسئول نگهداری اطلاعات موقعیت خود است.

۱۰-۱۰ کلاس‌های پروکسی

بخاطر دارید که دو جنبه و قاعده یک مهندسی نرم‌افزار ایده‌ال عبارت بودند از جداسازی واسط از پیاده‌سازی و پنهان‌سازی جزئیات پیاده‌سازی. برای برآورده کردن این اهداف سعی می‌کنیم تا کلاس را در یک فایل سرآیند تعریف کنیم و توابع عضو آنرا در یک فایل پیاده‌سازی کننده دیگر پیاده نمائیم. با این همه، همانطوری که در فصل نهم اشاره کردیم، فایلهای سرآیند حاوی بخش‌های از پیاده‌سازی کلاس هستند و تا حدودی به دیگران هم اشاره دارند. برای مثال اعضای خصوصی یک کلاس در تعریف کلاس در فایل سرآیند لیست می‌شوند، از اینرو این اعضا در دید سرویس‌گیرندگان قرار دارند، حتی اگر سرویس‌گیرنده‌ها به اعضای خصوصی دسترسی نداشته باشند. آشکار کردن داده خصوصی یک کلاس به این روش سبب می‌شود تا اطلاعات اختصاصی در معرض دید سرویس‌گیرنده‌های کلاس قرار گیرد. حال به معرفی نظریه کلاس پروکسی می‌پردازیم که اجازه می‌دهد تا حتی داده‌های خصوصی کلاس را از دید سرویس‌گیرنده‌های کلاس پنهان سازیم. مشروط بر اینکه سرویس‌گیرنده‌های کلاس با یک کلاس پروکسی مطلع باشند که فقط واسط سراسری به کلاس قادر به ارائه سرویس به سرویس گیرنده‌ها است، بدون اینکه سرویس‌گیرنده‌ها قادر به دسترسی به جزئیات پیاده سازی کلاس باشند.

پیاده‌سازی یک کلاس پروکسی مستلزم چندین مرحله است که در شکل‌های ۲۴-۱۰ الی ۲۷-۱۰ با مثال بیان شده است. ابتدا، تعریف کلاس را انجام می‌دهیم که حاوی پیاده‌سازی اختصاصی است و مایل هستیم تا آنرا پنهان نگه داریم. کلاس ما در این مثال، **Implementation** نام دارد و در شکل ۲۴-۱۰ نشان داده شده است. کلاس پروکسی **Interface** در شکل‌های ۲۵-۱۰ و ۲۶-۱۰ آورده شده است. برنامه تست و خروجی نمونه در شکل ۲۷-۱۰ دیده می‌شود.

کلاس **Implementation** (شکل ۲۴-۱۰) حاوی یک عضو داده خصوصی بنام value (داده‌ای که می‌خواهیم آنرا از دید سرویس‌گیرنده‌ها پنهان سازیم)، یک سازنده برای مقداردهی اولیه **value** و توابع **setValue** و **getValue** است.

یک کلاس پروکسی بنام **Interface** (شکل ۲۵-۱۰) با یک واسط سراسری **public** مشابه (بجز در اسامی سازنده و نابودکننده) برای کلاس **Implementation** تعریف کرده‌ایم.

```
1  // Fig. 10.24: Implementation.h
2  // Header file for class Implementation
3
4  class Implementation
5  {
6  public:
7     // constructor
8     Implementation( int v )
9        : value( v ) // initialize value with v
10    {
11       // empty body
12    } // end constructor Implementation
13
14    // set value to v
15    void setValue( int v )
16    {
17       value = v; // should validate v
18    } // end function setValue
19
20    // return value
21    int getValue() const
22    {
23       return value;
24    } // end function getValue
25 private:
26    int value; // data that we would like to hide from the client
27 }; // end class Implementation
```

شکل ۲۴-۱۰ | تعریف کلاس Implementation.

```
1  // Fig. 10.25: Interface.h
2  // Header file for class Interface
3  // Client sees this source code, but the source code does not reveal
4  // the data layout of class Implementation.
5
6  class Implementation; // forward class declaration required by line 17
7
8  class Interface
9  {
10 public:
11    Interface( int ); // constructor
12    void setValue( int ); // same public interface as
13    int getValue() const; // class Implementation has
14    ~Interface(); // destructor
15 private:
16    // requires previous forward declaration (line 6)
17    Implementation *ptr;
18 }; // end class Interface
```

شکل ۲۵-۱۰ | تعریف کلاس Interface.

تنها عضو خصوص کلاس پروکسی یک اشاره‌گر به شی از کلاس **Implementation** است. با استفاده از اشاره‌گر به این روش می‌توانیم جزئیات پیاده‌سازی کلاس **Implementation** را از دید سرویس‌گیرنده پنهان نگه داریم. توجه کنید که تنها اشاره‌ای که در کلاس **Interface** به کلاس اختصاصی

Implementation شده است، اعلان اشاره‌گر (خط 17) و در خط 6، اعلان رو به جلو کلاس است. زمانیکه تعریف کلاس (همانند کلاس **Interface**) فقط از یک اشاره‌گر یا مراجعه به یک شی از کلاس دیگری استفاده می‌کند (همانند یک شی از کلاس **Implementation**)، فایل سرآیند برای سایر کلاس‌ها، نیازی ندارد تا همراه با **#indude** باشد. می‌توانید به آسانی آنرا بعنوان یک نوع داده برای سایر کلاس‌ها توسط یک اعلان رو به جلو کلاس (*forward class declaration*) اعلان کنید (همانند خط 6) فایل پیاده‌سازی تابع عضو برای کلاس پروکسی **Interface** (شکل ۲۶-۱۰) تنها فایلی است که شامل فایل سرآیند **Implementation.h** (خط 5) حاوی کلاس **Implementation** می‌باشد. فایل **Interface.cpp** (شکل ۲۶-۱۰) بصورت یک فایل کد کامپایل شده به همراه فایل سرآیند **Interface.h** که حاوی نمونه‌های اولیه از سرویس‌های تدارک دیده شده توسط کلاس پروکسی است، در اختیار سرویس‌گیرنده گذاشته می‌شود. بدلیل اینکه فایل **Interface.cpp** فقط بصورت کد شی در اختیار سرویس‌گیرنده قرار دارد، سرویس‌گیرنده قادر به مشاهده تعامل‌های صورت گرفته مابین کلاس پروکسی و کلاس اختصاصی (خطوط 23، 17، 9 و 29) نخواهد بود. دقت کنید که کلاس پروکسی یک لایه اضافی به فراخوانی تابع اضافه می‌کند که هزینه پنهان‌سازی داده خصوص کلاس **Implementation** است. با توجه به سرعت روز افزون کامپیوترها و قابلیت کامپایلرها در فراخوانی اتوماتیک inline توابع، تاثیر این لایه در کارایی قابل چشم پوشی است.

```
1  // Fig. 10.26: Interface.cpp
2  // Implementation of class Interface--client receives this file only
3  // as precompiled object code, keeping the implementation hidden.
4  #include "Interface.h" // Interface class definition
5  #include "Implementation.h" // Implementation class definition
6
7  // constructor
8  Interface::Interface( int v )
9     : ptr ( new Implementation( v ) ) // initialize ptr to point to
10 {                                    // a new Implementation object
11    // empty body
12 } // end Interface constructor
13
14 // call Implementation's setValue function
15 void Interface::setValue( int v )
16 {
17    ptr->setValue( v );
18 } // end function setValue
19
20 // call Implementation's getValue function
21 int Interface::getValue() const
22 {
23    return ptr->getValue();
24 } // end function getValue
25
26 // destructor
27 Interface::~Interface()
28 {
29    delete ptr;
30 } // end ~Interface destructor
```

شکل ۲۶-۱۰ | تعریف تابع عضو کلاس Interface.

شکل ۲۷-۱۰ کلاس **Interface** را تست می‌کنید. دقت کنید که فقط فایل سرآیند برای **Interface** در کد سرویس گیرنده شامل شده است (خط 7)، در اینجا هیچ ذکری از وجود یک کلاس مجزا بنام **Implementation** نیست. از اینرو، سرویس‌گیرنده هرگز داده خصوصی کلاس **Implementation** را نخواهد دید.

مهندسی نرم‌افزار

کلاس پروکسی کد سرویس‌گیرنده را از تغییرات پیاده‌سازی دور نگه می‌دارد.

```
1  // Fig. 10.27: fig10_27.cpp
2  // Hiding a class’s private data with a proxy class.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Interface.h" // Interface class definition
8
9  int main()
10 {
11    Interface i( 5 ); // create Interface object
12
13    cout << "Interface contains: " << i.getValue()
14       << " before setValue" << endl;
15
16    i.setValue( 10 );
17
18    cout << "Interface contains: " << i.getValue()
19       << " after setValue" << endl;
20    return 0;
21 } // end main
```

```
Interface contains: 5 before setValue
Interface contains: 10 after setValue
```

شکل ۲۷-۱۰ | پیاده‌سازی کلاس پروکسی.

خود آزمایی

۱-۱۰ جاهای خالی را با کلمات مناسب پر کنید.

a) بایستی از ______ برای مقداردهی اولیه عضوهای ثابت یک کلاس استفاده کرد.

b) یک تابع غیر عضو باید بصورت یک ______ برای کلاسی اعلان شود که دارای دسترسی به اعضای داده خصوصی کلاس است.

c) عملگر ______ بصورت اتوماتیک حافظه دینامیکی برای شی از نوع مشخص شده اخذ کرده و یک ______ به آن نوع برگشت می‌دهد.

d) یک شی ثابت بایستی ______ شود، پس از ایجاد شدن مقدار آن قابل تغییر نخواهد بود.

e) یک عضو داده ______ نشانه اطلاعات در سطح کلاس است.

f) توابع عضو غیر استاتیک دارای دسترسی اشاره‌گر به خود هستند که اشاره‌گر ______ نامیده می‌شود.

g) کلمه کلیدی _____ مشخص می‌کند که یک شی یا متغیر پس از مقداردهی آن دیگر قابل تغییر دادن نمی‌باشد.

h) اگر یک عضو مقداردهی کننده برای یک شی عضو از کلاسی تدارک دیده نشده باشد، _____ برای آن شی فراخوانی خواهد شد.

i) یک تابع عضو بایستی بصورت static اعلان شود اگر دارای دسترسی _____ به اعضای کلاس باشد.

j) شی‌های عضو _____ از شی کلاس احاطه کننده ساخته می‌شوند.

k) عملگر _____ مبادرت به باز پس‌گیری حافظه اخذ شده توسط new می‌کند.

۲-۱۰ خطاهای موجود در کلاس زیر را یافته و آنها را اصلاح کنید:

```
class Example
{
 public :
       Example (int y = 10)
           :data (y)
       {
        //empty bady
       } //end Example constructor

       int getIncrementedData( ) const
       {
        return data++;
       } //end function getIncrementedData

       static int getCount( )
       {
        cout<< "Data is"<<data << endll;
        return count;
       } //end function get count
       private :
           int data :
           static int count :
       } //end class Example
```

پاسخ خودآزمایی

۱-۱۰) a) مقداردهی کننده عضو. b) دوست (friend) c) new، اشاره‌گر. d) مقداردهی اولیه. e) استاتیک f) this. g) h const) سازنده پیش فرض. f) غیر استاتیک. j) قبل. k) delete.

۲-۱۰)

خطا: تعریف کلاس Example دارای دو خطا است . اولین خطا در تابع getIneremented Data رخ داده است. تابع بصورت const اعلان شده، اما شی را دچار تغییر می‌سازد.

اصلاح: برای اصلاح اولین خطا، کلمه کلیدی const را از تعریف getIncrementedData حذف کنید.

خطا: دومین خطا در تابع getCount رخ داده است. این تابع بصورت استاتیک اعلان شده است، از اینرو اجازه ندارد تا به هر عضو غیر استاتیک کلاس دسترسی پیدا کند.

اصلاح: برای اصلاح دومین خطا، خط خروجی را از تعریف getCount حذف کنید.

تمرینات

۱۰-۳ به مقایسه اخذ حافظه دینامیکی و باز پس‌گیری آن توسط عملگرهای new، new[]، delete و delete[] بپردازید.

۱۰-۴ مفهوم دوستی را در C++ توضیح دهید. به بررسی جنبه‌های منفی رابطه دوستی هم بپردازید.

۱۰-۵ آیا می‌توان تعریف کلاس Time را به نحوی اصلاح کرد که در برگیرنده هر دو سازنده زیر باشد؟ اگر پاسخ منفی است، توضیح دهید چرا نمی‌توان اینکار را انجام داد.

```
Time ( int h = 0, int m = 0, int s= 0 ) ;
Time();
```

۱۰-۶ در صورتی‌که نوع برگشتی حتی از نوع void برای یک سازنده یا نابود کننده مشخص شود، چه اتفاقی خواهد افتاد؟

۱۰-۷ کلاس Date بکار رفته در شکل ۱۰-۱۰ را برای داشتن قابلیت‌های زیر تغییر دهید:

a) خروجی تاریخ در فرمت‌های مختلف همانند

DDD YYYY
MM/DD/YY
June 14,1992

b) استفاده از سازنده‌های سربارگذاری شده برای ایجاد شی‌های Date مقداردهی شده با فرمت‌های تاریخی مطرح شده در بخش (a).

c) ایجاد سازنده Date که تاریخ سیستم را با استفاده از توابع کتابخانه استاندارد <ctime> خوانده و اعضای Date را تنظیم کند.

۱۰-۸ کلاس SavingsAccaunt را ایجاد کنید. از یک عضو داده استاتیکی بنام annualInterestRate برای نرخ سود سالیانه برای هر سپرده استفاده کنید. هر عضو از کلاس حاوی یک عضو داده private بنام savingsBalance است که دلالت بر میزان پس انداز جاری در سپرده دارد. تابع عضو calculateMonthlyInterest را در نظر بگیرید که سود ماهانه را با ضرب موجودی (balance) در annualInterestRate تقسیم بر 12 بدست می‌آورد، این سود باید به savingsBalance افزوده شود. یک تابع عضو استاتیک بنام modifyInterestRate در نظر بگیرید که مقدار annualInterestRate را با یک مقدار جدید تنظیم کند. برنامه راه اندازی برای تست کلاس SavingsAccount بنویسید. دو شی نمونه سازی شده از کلاس SavingsAccount بنام‌های saver2 و saver1 ایجاد کنید. موجودی saver1 را با $2000.00 و saver2 را با $3000.00 تنظیم کنید. مقدار annualInterestRate را با 3 درصد تنظیم نمائید. سپس نرخ سود ماهانه را محاسبه و موجودی جدید را برای هر پس‌انداز چاپ کنید. سپس نرخ سود سالانه annualInterestRate را با 4 درصد تنظیم کرده، سود ماه بعد را محاسبه و میزان موجودی‌های جدید را چاپ نمائید.

فصل
یازدهم

سربارگذاری عملگر، رشته‌ها و آرایه‌ها

اهداف

- سربارگذاری عملگر چیست و چگونه می‌تواند برنامه‌ها را خواناتر و برنامه‌نویسی را راحتر کند.
- تعریف مجدد (سربارگذاری) عملگرها برای کار با کلاس‌های تعریف شده توسط کاربر.
- تفاوت مابین عملگرهای سربارگذاری شده باینری و غیرباینری.
- تبدیل شی‌ها از یک کلاس به کلاس دیگر.
- زمان سربارگذاری عملگرها.
- ایجاد کلاس‌های PhoneNumber، Array، String و Date برای توصیف سربارگذاری عملگر.

- **استفاده از عملگرهای سربارگذاری شده و توابع عضو از کلاس کتابخانه استاندارد String.**
- **استفاده از کلمه کلیدی Explicit.**

رئوس مطالب

۱۱-۱ مقدمه

در فصل‌های ۱۰-۹ به معرفی اصول اولیه کلاس‌ها در C++ پرداختیم. سرویس‌ها از طریق ارسال پیغام (بشکل فراخوانی توابع عضو) به شی‌ها دریافت می‌شوند. این نحوه فراخوانی تابع بر روی انواع خاصی از کلاس‌ها (همانند کلاس‌های محاسباتی) کار پر زحمتی است. همچنین، برخی از دستکاری‌ها رایج به کمک عملگرها صورت می‌گیرند (همانند ورودی و خروجی). برای انجام چنین اعمالی می‌توانیم از عملگرهای توکار C++ استفاده کنیم.

این فصل نشان می‌دهد که چگونه عملگرهای C++ می‌تواند با شی‌ها کار کنند، عملی که معروف به *سربارگذاری عملگر* است. این روش یک روش سر راست و طبیعی برای بسط C++ با این قابلیت‌های جدید است، اما اینکار بایستی با احتیاط صورت گیرد.

یک مثال از عملگر سربارگذاری شده در C++، عملگر >> است که هم بعنوان عملگر درج و هم بعنوان عملگر بیتی شیفت به‌چپ بکار گرفته می‌شود. به همین ترتیب، عملگر << می‌تواند سربارگذاری

گردد و بعنوان عملگر استخراج و هم بعنوان عملگر بیتی شیفت به راست بکار گرفته شود. هر دو این عملگرها در کتابخانه استاندارد C++ سربارگذاری شده‌اند.

اگرچه سربارگذاری عملگر قابلیتی نامتعارف بنظر می‌رسد، اما اکثر برنامه‌نویسان بصورت ضمنی و مرتباً از آن استفاده می‌کنند. برای مثال، خود زبان C++ مبادرت به سربارگذاری عملگر جمع (+) و عملگر تفریق (-) کرده است. این عملگرها براساس متن می‌توانند انجام دهنده محاسبه صحیح، اعشاری و اشاره‌گر باشند. برخی از عملگر به دفعات سربارگذاری می‌شوند، بویژه عملگر تخصیص و برخی از عملگرهای محاسباتی همانند + و -. کاری که عملگرهای سربارگذاری شده می‌توانند انجام دهند، توسط فراخوانی صریح توابع قابل اجرا است، اما نشان‌گذاری عملگر از وضوح بیشتری برخوردار بوده و برای برنامه‌نویسان آشناتر هستند.

برای توصیف نحوه سربارگذاری عملگرها، مبادرت به ایجاد کلاس‌های **PhoneNumber**، **Array**، **String** و **Date**، شامل عملگرهای درج، استخراج، تخصیص، تساوی، رابطه‌ای، شاخص، نفی منطقی، پرانتز و عملگرهای افزاینده خواهیم کرد. فصل با مثالی از کلاس **String** از کتابخانه استاندارد C++ که حاوی تعدادی عملگر سربارگذاری شده خاتمه می‌یابد.

۱۱-۲ اصول سربارگذاری عملگر

برنامه‌نویسی C++ یک فرآیند حساس به نوع و متمرکز بر نوع است. برنامه‌نویسان می‌توانند از نوع‌های بنیادین استفاده کرده و نوع‌های جدیدی تعریف کنند. نوع‌های بنیادین قادر به استفاده از انواع عملگرهای C++ هستند. عملگرهای تدارک دیده شده توسط برنامه‌نویسان با نشانه‌گذاری مختصر در عباراتی بکار می‌روند که دارای شی‌های از نوع‌های بنیادین می‌باشند.

بعلاوه برنامه‌نویسان می‌تواند از عملگرها به همراه نوع‌های تعریف شده از سوی کاربر کار کنند. اگرچه C++ اجازه ایجاد عملگرهای جدید را نمی‌دهد، اما اجازه می‌دهد تا اکثر عملگرهای موجود را به هنگام کار بر روی شی‌ها سربارگذاری کرد که خود قابلیتی توانمند است.

یک عملگر با نوشتن تعریف تابع عضو غیراستاتیک یا تعریف تابع سراسری سربارگذاری می‌شود، بجز اینکه نام تابع همراه با کلمه کلیدی **operator** و بدنبال آن سمبل عملگری که می‌خواهیم سربارگذاری شود، آورده می‌شود. برای مثال، نام تابع **operator+** می‌تواند برای سربارگذاری کردن عملگر جمع (+) بکار گرفته شود. زمانیکه عملگرها بعنوان تابع عضو سربارگذاری می‌شوند، بایستی غیراستاتیک باشند، چرا که باید بر روی یک شی از کلاس فراخوانی شده و بر روی آن شی عمل نمایند.

برای استفاده از یک عملگر بر روی شی‌های کلاس، آن عملگر باید سربارگذاری شده باشد، البته با سه استثناء. عملگر تخصیص (=) می‌تواند با هر کلاسی به منظور انجام تخصیص اعضای داده کلاس بکار گرفته شود – هر عضو داده از شی «منبع» به شی «هدف» تخصیص می‌یابد.

بزودی شاهد خواهید بود که چنین تخصیص پیش‌فرضی بر روی کلاس‌هائی با اعضای اشاره‌گر کار خطرناکی است. عملگرهای آدرس (&) و کاما (,) نیز می‌توانند با شی‌های هر کلاسی بکار گرفته شوند، بدون اینکه سربارگذاری شده باشند. عملگر آدرس، مبادرت به بازگرداندن آدرس شی از حافظه می‌کند. عملگر کاما مبادرت به ارزیابی عبارت از سمت چپ کرده، سپس از سمت راست می‌نماید. هر دو این عملگرها می‌توانند سربارگذاری شوند.

سربارگذاری فرآیند بسیار مناسبی برای کلاس‌های محاسباتی (ریاضی) است. انجام اینکار مستلزم سربارگذاری مجموعه‌ای از عملگرها است تا از عملکرد دقیق چنین کلاس‌های که در کارهای واقعی بکار گرفته می‌شوند، مطمئن گردیم. برای مثال، فقط سربارگذاری کردن عملگر جمع در یک کلاس از اعداد مختلط کار غیرعادی است، چرا که در اعداد مختلط از سایر عملگرهای ریاضی استفاده می‌شود.

سربارگذاری کردن عملگر همان عبارات کوتاه و آشنا را برای نوع تعریف شده توسط کاربر را فراهم می‌آورد که C++ با مجموعه‌ای غنی از عملگرهای خود برای نوع‌های بنیادین تدارک دیده است. سربارگذاری کردن عملگر یک فرآیند اتوماتیک نیست و بایستی توابع سربارگذاری عملگر را برای انجام مقاصد خود بنویسید. گاهی اوقات چنین توابعی می‌توانند بصورت توابع عضو، توابع **friend**، گاهی بصورت توابع سراسری و غیردوست ایجاد شوند. در این فصل به چنین مباحثی خواهیم پرداخت.

۱۱-۳ محدودیت‌های سربارگذاری عملگر

اکثر عملگرهای C++ قادر به سربارگذاری شدن هستند. این عملگرها در جدول شکل ۱۱-۱ نشان داده شده‌اند. در جدول شکل ۱۱-۲ عملگرهای که نمی‌توانند سربارگذاری شوند، لیست شده‌اند.

خطای برنامه‌نویسی

مبادرت به سربارگذاری کردن یک عملگر که نمی‌تواند سربارگذاری شود، خطای نحوی است.

عملگرهای قابل سربارگذاری							
+	-	*	/	%	^	&	|
~	!	=	<	>	+=	-=	*=
/=	%=	^=	&=	|=	<<	>>	>>=

<<=	==	!=	<=	>=	&&	||	++
--	->*	,	->	[]	()	new	delete
new[]	delete[]						

شکل ۱-۱۱ | عملگرهای که می‌توانند سربارگذاری شوند.

عملگرهای غیرقابل سربارگذاری			
.	.*	::	?:

شکل ۲-۱۱ | عملگرهای که نمی‌توانند سربارگذاری شوند.

تقدم، شرکت‌پذیری و تعداد عملوند

تقدم یا اولویت یک عملگر را نمی‌توان با سربارگذاری تغییر داد. چنین عملی می‌تواند شرایط ناخواسته‌ای را سبب شود. با این همه، می‌توان از پرانتزها استفاده کرده و ترتیب ارزیابی عملگرهای سربارگذاری شده در یک عبارت را بدست گرفت.

شرکت‌پذیری یک عملگر (یعنی اعمال عملگر از راست به چپ یا از چپ به راست) را نمی‌توان با سربارگذاری کردن تغییر داد. امکان تغییر در تعداد عملوندهای که یک عملگر می‌تواند برای آنها اثر کند وجود ندارد.

ایجاد عملگرهای جدید

امکان ایجاد عملگر جدید وجود ندارد، فقط می‌توان عملگرهای موجود را سربارگذاری کرد. متاسفانه، چنین رفتاری سبب می‌شود تا برنامه‌نویس قادر به استفاده از نمادهای رایجی همانند عملگر ** که در سایر زبان‌های برنامه‌نویسی برای توان بکار گرفته می‌شود، نباشد [نکته: می‌توانید عملگر ^ را برای انجام توان سربارگذاری نمائید، که در برخی از زبان‌ها کاربرد دارد.]

خطای برنامه‌نویسی

اقدام به ایجاد عملگرهای جدید از طریق سربارگذاری عملگر، یک خطای نحوی است.

عملگرها با نوع‌های بنیادین

مفهوم و معنی نحوه عملکرد یک عملگر بر روی شی‌های از نوع‌های بنیادین را نمی‌توان با سربارگذاری عملگر تغییر داد. برای مثال، برنامه‌نویس نمی‌تواند مفهوم افزودن یا جمع دو مقدار صحیح را تغییر دهد. سربارگذاری عملگر فقط با شی‌های از نوع‌های تعریف شده از سوی کاربر یا ترکیبی از یک شی از نوع تعریف شده از سوی کاربر و یک شی از نوع بنیادین کار می‌کند.

عملگرهای وابسته

سربارگذاری یک عملگر تخصیص و یک عملگر جمع به عبارتی همانند عبارت زیر اجازه می‌دهد

```
object2 = object2 + object+1;
```

به این مفهوم نیست که عملگر =+ هم سربارگذاری شده است و عبارتی مانند عبارت زیر داشت

```
object2 += object+1;
```

چنین رفتاری فقط با اعلان صریح سربارگذاری عملگر =+ برای آن کلاس صورت می‌گیرد.

خطای برنامه‌نویسی

فرض اینکه با سربارگذاری یک عملگر همانند +، سایر عملگرهای وابسته همانند =+ یا سربارگذاری == سبب سربارگذاری شدن عملگری مانند =! خواهد شد، خطا است. هر عملگر بایستی بصورت صریح سربارگذاری شود و سربارگذاری ضمنی وجود ندارد.

۱۱-٤ توابع عملگر بعنوان اعضای کلاس در مقابل توابع سراسری

توابع عملگر می‌توانند توابع عضو یا توابع سراسری باشند، غالباً توابع سراسری بدلیل کارایی بصورت دوست (friend) ایجاد می‌شوند. توابع عضو از اشاره‌گر **this** بصورت ضمنی استفاده می‌کنند تا یکی از آرگومان‌های شی کلاس را بدست آورند (عملوند سمت چپ در عملگرهای باینری) آرگومان‌ها برای هر دو عملوند در یک عملگر باینری بایستی بصورت صریح در فراخوانی یک تابع سراسری لیست شده باشند.

عملگرهای که باید بعنوان توابع عضو سربارگذاری شوند

به هنگام سربارگذاری **()**، **[]**، <- یا هر عملگر تخصیصی، عملگر سربارگذاری کننده تابع باید بصورت یک عضو کلاس اعلان شود. برای سایر عملگرها توابع سربارگذاری توابع می‌توانند اعضای کلاس یا توابع سراسری باشند.

عملگرها بعنوان توابع عضو و توابع سراسری

خواه یک تابع عملگر بصورت یک تابع عضو یا یک تابع سراسری پیاده‌سازی شده باشد، عملگر هنوز هم به همان روش در عبارات بکار گرفته می‌شود. بنابر این کدام روش پیاده‌سازی بهتر است؟

زمانیکه یک تابع عملگر بصورت یک تابع عضو پیاده‌سازی می‌شود، سمت چپ‌ترین عملوند بایستی یک شی (با یک مراجعه به یک شی) از کلاس عملگر باشد. اگر عملوند سمت چپ باید شی از یک کلاس متفاوت یا یک نوع بنیادین باشد، این تابع عملگر بایستی بصورت یک تابع سراسری پیاده‌سازی

شود. یک تابع عملگر سراسری می‌تواند بصورت یک **friend** از یک کلاس ایجاد شود اگر آن تابع بصورت مستقیم به اعضای **private** یا **protected** آن کلاس دسترسی دارد.

توابع عضو عملگر از یک کلاس خاص فقط در صورتیکه عملوند سمت چپ یک عملگر باینری، یک شی از آن کلاس باشد، یا زمانیکه عملوند منفرد از یک عملگر غیرباینری، یک شی از آن کلاس باشد، توسط کامپایلر فراخوانی خواهد شد (بصورت ضمنی).

چرا سربارگذاری عملگرهای درج و استخراج بصورت توابع سراسری سربارگذاری می‌شوند

عملگر درج جریان **(<<)** سربارگذاری شده در عبارتی که عملوند سمت چپ آن دارای نوع **ostream &** بصورت **cout<<classObject** است، بکار گرفته می‌شود. برای استفاده از عملگر به این روش که در آن عملوند از سمت راست، یک شی از یک کلاس تعریف شده از سوی کاربر است، بایستی بصورت یک تابع سراسری سربارگذاری شود. برای یک تابع عضو، عملگر << مجبور است تا بصورت عضوی از کلاس **ostream** باشد. اینکار برای کلاس‌های تعریف شده توسط کاربر امکان‌پذیر نیست، از آنجا که اجازه نداریم تا کلاس‌های کتابخانه استاندارد C++ را تغییر دهیم. به همین ترتیب از عملگر استخراج جریان (>>) سربارگذاری شده در عبارتی که عملوند سمت چپ دارای نوع **istream &**، بصورت **cin>>classObject** است و عملوند سمت راست یک شی از یک کلاس تعریف شد توسط کاربر است، بایستی بصورت یک تابع سراسری باشد. همچنین امکان دارد هر یک از این توابع عملگر سربارگذاری شده نیاز به دسترسی به اعضای داده **private** داشته باشند، از اینرو چنین توابعی می‌توانند بصورت توابع **friend** کلاس ایجاد شوند تا کارایی افزایش یابد.

جابجایی عملگرها

یکی دیگر از دلایل انتخاب توابع سراسری برای سربارگذاری یک عملگر امکان جابجا کردن عملگر است. برای مثال، فرض کنید یک شی **number** از نوع **long int** داریم و یک شی بنام **bigInteger1** از کلاس **HugeInteger** (کلاسی که تعداد ارقام بیش از ظرفیت سایز word در کامپیوتر است). عملگر جمع (+) بطور موقت یک شی **HugeInteger** بعنوان مجموع یک **HugeInteger** و یک **long int** (در عبارتی بصورت **bigInteger1 + number**)، یا بعنوان مجموع یک **long int** و یک **HugeInteger** (در عبارتی بصورت **number + bigIneger1**) تولید می‌کند. از اینرو، نیازمند عملگر جمعی هستیم که جابجاپذیر باشد. مشکل اینجاست که شی کلاس باید در سمت چپ عملگر جمع قرار گیرد، اگر آن عملگر بعنوان یک تابع عضو سربارگذاری شده باشد. از اینرو، اقدام به سربارگذاری عملگر بفرم یک تابع

سراسری می‌کنیم تا به **HugeInteger** اجازه دهد تا در سمت راست جمع قرار داده شود. تابع **operator+** که با **HugeInteger** در سمت چپ کار می کند، هنوز هم می‌تواند یک تابع عضو باشد.

۱۱-۵ سربارگذاری عملگرهای درج و استخراج

زبان C++ قادر است تا با استفاده از عملگرهای درج (<<) و استخراج و استخراج (>>) مبادرت به ورود و خروج نوع‌های بنیادین کند. کتابخانه‌های کلاس تدارک دیده شده همراه کامپایلرهای C++ مبادرت به سربارگذاری این عملگرها برای پردازش هر نوع بنیادین می‌کنند که شامل اشاره‌گرها و رشته‌های Char * هم می‌شود. همچنین می‌توان برای انجام عملیات ورودی و خروجی بر روی نوع‌های تعریف شده توسط کاربر مبادرت به سربارگذاری عملگرهای درج و استخراج کرد. برنامه موجود در شکل‌های ۳-۱۱ الی ۵-۱۱ به توصیف نحوه سربارگذاری این عملگرها برای رسیدگی به داده تعریف شده از سوی کاربر که یک شماره تلفن است و در کلاسی بنام **PhoneNumber** قرارداد، می‌پردازد. فرض برنامه بر این است که شماره تلفن‌ها بدرستی وارد شده‌اند.

```
1  // Fig. 11.3: PhoneNumber.h
2  // PhoneNumber class definition
3  #ifndef PHONENUMBER_H
4  #define PHONENUMBER_H
5
6  #include <iostream>
7  using std::ostream;
8  using std::istream;
9
10 #include <string>
11 using std::string;
12
13 class PhoneNumber
14 {
15    friend ostream &operator<<( ostream &, const PhoneNumber & );
16    friend istream &operator>>( istream &, PhoneNumber & );
17 private:
18    string areaCode; // 3-digit area code
19    string exchange; // 3-digit exchange
20    string line; // 4-digit line
21 }; // end class PhoneNumber
22
23 #endif
```

شکل ۳-۱۱ | کلاس PhoneNumber با عملگرهای سربارگذاری شده درج و استخراج بعنوان توابع frirend.

```
1  // Fig. 11.4: PhoneNumber.cpp
2  // Overloaded stream insertion and stream extraction operators
3  // for class PhoneNumber.
4  #include <iomanip>
5  using std::setw;
6
7  #include "PhoneNumber.h"
8
9  // overloaded stream insertion operator; cannot be
10 // a member function if we would like to invoke it with
11 // cout << somePhoneNumber;
12 ostream &operator<<( ostream &output, const PhoneNumber &number )
13 {
```

```
14    output << "(" << number.areaCode << ") "
15       << number.exchange << "-" << number.line;
16    return output; // enables cout << a << b << c;
17 } // end function operator<<
18
19 // overloaded stream extraction operator; cannot be
20 // a member function if we would like to invoke it with
21 // cin >> somePhoneNumber;
22 istream &operator>>( istream &input, PhoneNumber &number )
23 {
24    input.ignore(); // skip (
25    input >> setw( 3 ) >> number.areaCode; // input area code
26    input.ignore( 2 ); // skip ) and space
27    input >> setw( 3 ) >> number.exchange; // input exchange
28    input.ignore(); // skip dash (-)
29    input >> setw( 4 ) >> number.line; // input line
30    return input; // enables cin >> a >> b >> c;
31 } // end function operator>>
```

شکل ٤-١١ | سربارگذاری عملگرهای درج و استخراج برای کلاس PhoneNumber.

```
1  // Fig. 11.5: fig11_05.cpp
2  // Demonstrating class PhoneNumber's overloaded stream insertion
3  // and stream extraction operators.
4  #include <iostream>
5  using std::cout;
6  using std::cin;
7  using std::endl;
8
9  #include "PhoneNumber.h"
10
11 int main()
12 {
13    PhoneNumber phone; // create object phone
14
15    cout << "Enter phone number in the form (123) 456-7890:" << endl;
16
17    // cin >> phone invokes operator>> by implicitly issuing
18    // the global function call operator>>( cin, phone )
19    cin >> phone;
20
21    cout << "The phone number entered was: ";
22
23    // cout << phone invokes operator<< by implicitly issuing
24    // the global function call operator<<( cout, phone )
25    cout << phone << endl;
26    return 0;
27 } // end main
```

```
Enter phone number in the form (123) 456-7890:
(800) 555-1212
The phone number enterd was: (800) 555-1212
```

شکل ٥-١١ | سربارگذاری عملگرهای درج و استخراج.

تابع عملگر استخراج **operator>>** (شکل ۴-۱۱، خطوط 31-22) مبادرت به دریافت مراجعه **istream** به **input** و **PhoneNumber** به **num** بعنوان آرگومان کرده و یک مراجعه **istream** برگشت می‌دهد. تابع عملگر **operator>>** شماره تلفن را بفرم زیر دریافت می‌کند.

```
(800) 555-1212
```

و در کلاس **PhoneNumber** قرار می‌دهد. زمانیکه کامپایلر با عبارت زیر مواجه می‌شود (خط 19 از شکل ۵-۱۱)

```
cin >> phone
```

کامپایلر مبادرت به فراخوانی تابع سراسری زیر می‌کند

```
operator>>(cin,phone);
```

زمانیکه این فراخوانی اجرا می‌شود، پارامتر مراجعه **input** (شکل ۴-۱۱، خط 22) تبدیل به یک عبارت مستعار برای **cin** شده و پارامتر مراجعه **number** تبدیل به یک مستعار برای **phone** می‌گردد. تابع عملگر، رشته‌ها را بصورت سه بخشی از شماره تلفن خوانده و در اعضای **areaCode** (خط 25)، **exchange** (خط 27) و **line** (خط 29) از شی **PhoneNumber** وارد می‌کند که توسط پارامتر **number** مورد مراجعه قرار دارد. دستکاری کننده استریم **setw** محدود کننده تعداد کاراکترهای خوانده شده به هر آرایه کاراکتری است. زمانیکه با **cin** و رشته‌ها بکار گرفته می‌شود، setw تعداد کاراکترهای خوانده شده را محدود به تعداد کاراکترهای تعیین شده توسط آرگومان خود می کند (مثلاً **setw(3)** اجازه خواندن سه کاراکتر را می‌دهد). پرانتزها، فاصله‌ها و کاراکترهای خط تیره با فراخوانی تابع عضو **istream** نادیده گرفته می‌شوند (شکل ۴-۱۱، خطوط 26,24 و 28)، که تعداد مشخصی از کاراکترها در استریم ورودی به کنار گذاشته می‌شوند. تابع **operator**>> مراجعه ورودی را برگشت می‌دهد (یعنی **cin**). اینکار به عملیات ورودی بر روی شی‌های **PhoneNumber** امکان می‌دهد تا متصل با عملیات ورودی بر روی دیگر شی‌های **PhoneNumber** یا شی‌های از سایر نوع‌های داده، به پیش رود. برای مثال، برنامه می‌تواند دو شی **PhoneNumber** را در یک عبارت وارد کند، همانند

```
cin >> phone1 >> phone2;
```

ابتدا عبارت **cin**>>**phone1** با فراخوانی تابع سراسری

```
operator>>(cin, phone1);
```

اجرا می‌شود. سپس این فراخوانی یک مراجعه به **cin** بعنوان مقداری از **cin**>>**phone1** برگشت می‌دهد، از اینرو مابقی بخشی باقیمانده عبارت بصورت **cin**>>**phone2** ارزیابی می‌گردد. اینکار با فراخوانی تابع سراسری صورت می‌گیرد.

```
operator>>(cin, phone2);
```

تابع عملگر درج (شکل ۴-۱۱، خطوط 17-12) یک مراجعه **ostream** (خروجی– output) و یک مراجعه ثابت **PhoneNumber** بعنوان آرگومان دریافت و یک مراجعه **ostream** برگشت می‌دهد. تابع >> **opereator** شی‌های از نوع **PhoneNumber** برگشت می‌دهد. زمانیکه کامپایلر به عبارت زیر می‌رسد (خط 25 از شکل ۵-۱۱)

```
cout << phone
```

کامپایلر یک فراخوانی تابع سراسری را بوجود می‌آورد

```
operator<<(cout, phone);
```

تابع **operator>>** بخش‌های یک شماره تلفن را بصورت رشته‌های به نمایش در می‌آورد، چرا که آنها بعنوان شی‌های رشته ذخیره شده بودند.

توجه کنید که توابع **operator<<** و **operator>>** در **phoneNumber** بصورت سراسری، توابع دوست (*friend*) اعلان شده‌اند (شکل ۳-۱۱، خطوط 16-15). اینها توابع سراسری هستند چرا که شی از کلاس **PhoneNumber** در هر حالت بصورت عملوند سمت راست عملگر ظاهر می‌شود. بخاطر دارید که، توابع عملگر سربارگذاری شده برای عملگرهای باینری می‌تواند توابع عضو باشند. در صورتیکه فقط عملوند سمت چپ یک شی از کلاسی باشد که در آن تابع عضو باشد. اگر عملگرهای ورودی و خروجی سربارگذاری شده نیاز به دسترسی مستقیم به اعضای کلاس غیر *public* داشته باشند، می‌تواند بصورت **friend** اعلان شوند. اینکار می‌تواند به دلایل کارایی باشد یا اینکه کلاس حاوی توابع *get* مقتضی نباشد. همچنین دقت کنید که مراجعه **PhoneNumber** در لیست پارامتری **operator>>** (شکل ۴-۱۱، خط 12) ثابت (const) است، چرا که **PhoneNumber** فقط خروجی است و مراجعه **PhoneNumber** در لیست پارامتری **operator<<** (خط 22) یک مقدار غیرثابت است، به این دلیل که شی **phoneNumber** بایستی برای ذخیره‌سازی شماره تلفن در یک شی، تغییرپذیر باشد.

۶-۱۱ سربارگذاری عملگرهای غیرباینری

یک عملگر غیرباینری در ارتباط با یک کلاس می‌تواند بصورت یک تابع غیراستاتیک بدون آرگومان یا بصورت یک تابع سراسری با یک آرگومان، که آن آرگومان بایستی یک شی از کلاس یا مراجعه‌ای به شی از آن کلاس باشد، سربارگذاری شود. توابع عضو که عملگرهای سربارگذاری شده را پیاده‌سازی می‌کنند بایستی بصورت غیراستاتیک باشند، از اینروست که می‌توانند به داده غیراستاتیک در هر شی از کلاس دسترسی پیدا کنند. بخاطر داشته باشید که توابع عضو استاتیک فقط می‌توانند به اعضای داده استاتیک کلاس دسترسی پیدا کنند.

در ادامه این فصل مبادرت به سربارگذاری عملگر غیرباینری **!** برای تست این مطلب خواهیم کرد که آیا یک شی که از کلاس **String** ایجاد می‌کنیم (بخش ۱۰-۱۱) تهی بوده و نتیجه **bool** برگشت می‌دهد یا خیر. به عبارت **s!** توجه کنید که در آن **s** یک شی از کلاس **String** است. زمانیکه یک عملگر غیرباینری همانند **!** بصورت یک تابع عضو سربارگذاری می‌شود (بدون آرگومان) و کامپایلر عبارت **s!** را مشاهده می‌کند، مبادرت به فراخوانی **s.operator!()** می‌نماید. عملوند **s** شی از کلاس **String** است. تابع در تعریف کلاس بصورت زیر اعلان شده است:

```
class String
```

```
{
public:
      bool operator!() const;
}; //end class String
```

یک عملگر غیرباینری همانند **!** می‌توانند به دو روش به همراه یک آرگومان بصورت یک تابع سراسری سربارگذاری گردد، خواه با یک آرگومان که یک شی است (اینکار مستلزم یک کپی از شی بوده، از اینرو اثرات جانبی تاثیری بر شی واقعی نخواهند داشت) یا با آرگومانی که یک مراجعه به یک شی است (کپی از شی اصلی وجود ندارد، از اینرو تمام تاثیرات جانبی این تابع بر روی شی اصلی یا واقعی تاثیرگذار خواهند بود). اگر **s** یک شی از کلاس **String** باشد (یا یک مراجعه به یک شی از کلاس **String**)، پس با **!s** همانند فراخوانی **operator!(s)** رفتار خواهد شد که فراخوانی آن بصورت زیر اعلان شده است:

```
bool operator!(const String &);
```

۱۱-۷ سربارگذاری عملگرهای باینری

عملگر باینری می‌تواند بصورت یک تابع عضو غیراستاتیک با یک آرگومان یا بصورت یک تابع سراسری با دو آرگومان (یکی از این آرگومان‌ها باید یک شی کلاس یا یک مراجعه به شی کلاس باشد) سربارگذاری گردد.

در ادامه این فصل، مبادرت به سربارگذاری > برای مقایسه دو شی رشته‌ای خواهیم کرد. در زمان سربارگذاری عملگر باینری > بصورت یک تابع عضو غیراستاتیک از کلاس **String** با یک آرگومان، اگر **y** و **z** شی‌های از کلاس **String** باشند، پس با **y>z** بصورت **y.operator(z)** رفتار خواهد شد، که فراخوانی تابع عضو > **operator** بصورت زیر اعلان شده است.

```
class String
{
public:
      bool operator<(const String &) const;
}; //end class String
```

اگر عملگر باینری > بصورت یک تابع سراسری سربارگذاری شود، بایستی دو آرگومان دریافت کند. اگر **y** و **z** شی‌های از کلاس **String** باشند یا مراجعه‌ای به شی‌های از کلاس **String**، پس با **y>z** بصورت **operator(y,z)** رفتار خواهد شد، اعلان تابع سراسری > **operator** بصورت زیر است:

```
bool operator<(const String &, const Strig &);
```

۱۱-۸ مبحث آموزشی: کلاس Array

آرایه‌های مبتنی بر اشاره‌گر چندین مشکل دارند. برای مثال، برنامه می‌تواند به راحتی از مرزهای آرایه خارج شود، چرا که C++ تستی بر روی مرزهای آرایه انجام نمی‌دهد (خود برنامه‌نویس می‌تواند اینکار را انجام دهد). آرایه‌های با سایز n بایستی عناصری به تعداد *0, ..., n-1* داشته باشند، تغییر محدودهٔ شاخص

امکان‌پذیر نمی‌باشد. کل یک آرایه غیرکاراکتری را نمی‌توان به یکباره از ورودی دریافت یا در خروجی قرار داد، هر عنصر آرایه بایستی بصورت جداگانه خوانده یا نوشته شود. دو آرایه را نمی‌توان با عملگرهای تساوی یا رابطه‌ای و آن هم بصورت معنی‌دار مقایسه کرد (چرا که اسامی آرایه‌ها اشاره‌گرهایی به شروع آرایه در حافظه هستند و البته دو آرایه همیشه در دو مکان متفاوت حافظه خواهند بود). زمانیکه یک آرایه به یک تابع چندمنظوره طراحی شده برای کار با آرایه‌ها با هر سایزی ارسال می‌شود، سایز آرایه بایستی بعنوان یک آرگومان اضافی به تابع ارسال گردد. یک آرایه را نمی‌توان به آرایه دیگری با عملگر تخصیص، انتساب داد (چرا که اسامی آرایه‌ها اشاره‌گرهای ثابت (*const*) بوده و از اشاره‌گر ثابت نمی‌توان در سمت چپ یک عملگر تخصیص استفاده کرد). بنظر می‌رسد که انجام چنین کارهای با آرایه‌ها نبایستی کار غیرعادی باشد، اما آرایه‌های مبتنی بر اشاره‌گر دارای چنین قابلیت‌های نیستند. با این همه C++ مفهومی برای پیاده‌سازی آرایه‌ها با چنین قابلیت‌های از طریق استفاده از کلاس‌ها و سربارگذاری عملگر تدارک دیده است.

در این مثال، یک کلاس آرایه قدرتمند ایجاد خواهیم کرد که قادر به انجام تست بر روی مرزهای آرایه است. کلاس به یک شی آرایه اجازه می‌دهد تا با استفاده از عملگر تخصیص به یک آرایه دیگر انتساب یابد. شی‌ها از کلاس **Array** از سایز خود مطلع بوده و از اینرو نیازی نیست تا سایز آرایه بعنوان یک آرگومان مجزا به هنگام ارسال آرایه به یک تابع همراه شود. کل آرایه را می‌توان با استفاده از عملگرهای درج و استخراج از ورودی دریافت و در خروجی قرار داد. مقایسه آرایه را می‌توانیم با عملگرهای تساوی== و =! انجام دهیم.

این مثال درک شما را از انتزاعی کردن داده افزایش خواهد داد. امکان دارد که بخواهید قابلیت‌های دیگری به این کلاس آرایه اضافه کنید. برنامه موجود در شکل‌های ۶-۱۱ الی ۸-۱۱ به توصیف کلاس **Array** و سربارگذاری عملگرهای آن می‌پردازد. ابتدا به سراغ **main** می‌رویم (شکل ۸-۱۱). سپس به تعریف کلاس (شکل ۶-۱۱) و هر یک از تعاریف توابع عضو و توابع **friend** کلاس می‌پردازیم (شکل ۷-۱۱).

```
1  // Fig. 11.6: Array.h
2  // Array class for storing arrays of integers.
3  #ifndef ARRAY_H
4  #define ARRAY_H
5
6  #include <iostream>
7  using std::ostream;
8  using std::istream;
9
10 class Array
11 {
12    friend ostream &operator<<( ostream &, const Array & );
13    friend istream &operator>>( istream &, Array & );
14 public:
15    Array( int = 10 ); // default constructor
```

```
16    Array( const Array & ); // copy constructor
17    ~Array(); // destructor
18    int getSize() const; // return size
19
20    const Array &operator=( const Array & ); // assignment operator
21    bool operator==( const Array & ) const; // equality operator
22
23    // inequality operator; returns opposite of == operator
24    bool operator!=( const Array &right ) const
25    {
26       return ! ( *this == right ); // invokes Array::operator==
27    } // end function operator!=
28
29    // subscript operator for non-const objects returns modifiable lvalue
30    int &operator[]( int );
31
32    // subscript operator for const objects returns rvalue
33    int operator[]( int ) const;
34 private:
35    int size; // pointer-based array size
36    int *ptr; // pointer to first element of pointer-based array
37 }; // end class Array
38
39 #endif
```

شکل ٦-١١ | تعریف کلاس Array با عملگرهای سربارگذاری شده.

```
1   // Fig 11.7: Array.cpp
2   // Member-function definitions for class Array
3   #include <iostream>
4   using std::cerr;
5   using std::cout;
6   using std::cin;
7   using std::endl;
8
9   #include <iomanip>
10  using std::setw;
11
12  #include <cstdlib> // exit function prototype
13  using std::exit;
14
15  #include "Array.h" // Array class definition
16
17  // default constructor for class Array (default size 10)
18  Array::Array( int arraySize )
19  {
20     size = ( arraySize > 0 ? arraySize : 10 ); // validate arraySize
21     ptr = new int[ size ]; // create space for pointer-based array
22
23     for ( int i = 0; i < size; i++ )
24        ptr[ i ] = 0; // set pointer-based array element
25  } // end Array default constructor
26
27  // copy constructor for class Array;
28  // must receive a reference to prevent infinite recursion
29  Array::Array( const Array &arrayToCopy )
30     : size( arrayToCopy.size )
31  {
32     ptr = new int[ size ]; // create space for pointer-based array
33
34     for ( int i = 0; i < size; i++ )
35        ptr[ i ] = arrayToCopy.ptr[ i ]; // copy into object
36  } // end Array copy constructor
37
38  // destructor for class Array
39  Array::~Array()
40  {
41     delete [] ptr; // release pointer-based array space
42  } // end destructor
43
44  // return number of elements of Array
```

```
45  int Array::getSize() const
46  {
47     return size; // number of elements in Array
48  } // end function getSize
49
50  // overloaded assignment operator;
51  // const return avoids: ( a1 = a2 ) = a3
52  const Array &Array::operator=( const Array &right )
53  {
54     if ( &right != this ) // avoid self-assignment
55     {
56        // for Arrays of different sizes, deallocate original
57        // left-side array, then allocate new left-side array
58        if ( size != right.size )
59        {
60           delete [] ptr; // release space
61           size = right.size; // resize this object
62           ptr = new int[ size ]; // create space for array copy
63        } // end inner if
64
65        for ( int i = 0; i < size; i++ )
66           ptr[ i ] = right.ptr[ i ]; // copy array into object
67     } // end outer if
68
69     return *this; // enables x = y = z, for example
70  } // end function operator=
71
72  // determine if two Arrays are equal and
73  // return true, otherwise return false
74  bool Array::operator==( const Array &right ) const
75  {
76     if ( size != right.size )
77        return false; // arrays of different number of elements
78
79     for ( int i = 0; i < size; i++ )
80        if ( ptr[ i ] != right.ptr[ i ] )
81           return false; // Array contents are not equal
82
83     return true; // Arrays are equal
84  } // end function operator==
85
86  // overloaded subscript operator for non-const Arrays;
87  // reference return creates a modifiable lvalue
88  int &Array::operator[]( int subscript )
89  {
90     // check for subscript out-of-range error
91     if ( subscript < 0 || subscript >= size )
92     {
93        cerr << "\nError: Subscript " << subscript
94           << " out of range" << endl;
95        exit( 1 ); // terminate program; subscript out of range
96     } // end if
97
98     return ptr[ subscript ]; // reference return
99  } // end function operator[]
100
101 // overloaded subscript operator for const Arrays
102 // const reference return creates an rvalue
103 int Array::operator[]( int subscript ) const
104 {
105    // check for subscript out-of-range error
106    if ( subscript < 0 || subscript >= size )
107    {
108       cerr << "\nError: Subscript " << subscript
109          << " out of range" << endl;
110       exit( 1 ); // terminate program; subscript out of range
111    } // end if
112
113    return ptr[ subscript ]; // returns copy of this element
114 } // end function operator[]
```

```
115
116 // overloaded input operator for class Array;
117 // inputs values for entire Array
118 istream &operator>>( istream &input, Array &a )
119 {
120    for ( int i = 0; i < a.size; i++ )
121       input >> a.ptr[ i ];
122
123    return input; // enables cin >> x >> y;
124 } // end function
125
126 // overloaded output operator for class Array
127 ostream &operator<<( ostream &output, const Array &a )
128 {
129    int i;
130
131    // output private ptr-based array
132    for ( i = 0; i < a.size; i++ )
133    {
134       output << setw( 12 ) << a.ptr[ i ];
135
136       if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
137          output << endl;
138    } // end for
139
140    if ( i % 4 != 0 ) // end last line of output
141       output << endl;
142
143    return output; // enables cout << x << y;
144 } // end function operator<<
```

شکل ۷-۱۱ | تعاریف عضو و friend کلاس Array.

```
1  // Fig. 11.8: fig11_08.cpp
2  // Array class test program.
3  #include <iostream>
4  using std::cout;
5  using std::cin;
6  using std::endl;
7
8  #include "Array.h"
9
10 int main()
11 {
12    Array integers1( 7 ); // seven-element Array
13    Array integers2; // 10-element Array by default
14
15    // print integers1 size and contents
16    cout << "Size of Array integers1 is "
17       << integers1.getSize()
18       << "\nArray after initialization:\n" << integers1;
19
20    // print integers2 size and contents
21    cout << "\nSize of Array integers2 is "
22       << integers2.getSize()
23       << "\nArray after initialization:\n" << integers2;
24
25    // input and print integers1 and integers2
26    cout << "\nEnter 17 integers:" << endl;
27    cin >> integers1 >> integers2;
28
29    cout << "\nAfter input, the Arrays contain:\n"
30       << "integers1:\n" << integers1
31       << "integers2:\n" << integers2;
32
33    // use overloaded inequality (!=) operator
34    cout << "\nEvaluating: integers1 != integers2" << endl;
35
36    if ( integers1 != integers2 )
37       cout << "integers1 and integers2 are not equal" << endl;
38
```

```
39    // create Array integers3 using integers1 as an
40    // initializer; print size and contents
41    Array integers3( integers1 ); // invokes copy constructor
42
43    cout << "\nSize of Array integers3 is "
44       << integers3.getSize()
45       << "\nArray after initialization:\n" << integers3;
46
47    // use overloaded assignment (=) operator
48    cout << "\nAssigning integers2 to integers1:" << endl;
49    integers1 = integers2; // note target Array is smaller
50
51    cout << "integers1:\n" << integers1
52       << "integers2:\n" << integers2;
53
54    // use overloaded equality (==) operator
55    cout << "\nEvaluating: integers1 == integers2" << endl;
56
57    if ( integers1 == integers2 )
58       cout << "integers1 and integers2 are equal" << endl;
59
60    // use overloaded subscript operator to create rvalue
61    cout << "\nintegers1[5] is " << integers1[ 5 ];
62
63    // use overloaded subscript operator to create lvalue
64    cout << "\n\nAssigning 1000 to integers1[5]" << endl;
65    integers1[ 5 ] = 1000;
66    cout << "integers1:\n" << integers1;
67
68    // attempt to use out-of-range subscript
69    cout << "\nAttempt to assign 1000 to integers1[15]" << endl;
70    integers1[ 15 ] = 1000; // ERROR: out of range
71    return 0;
72 } // end main
```

```
Size of Array integers1 is 7
Array after initialization:
           0           0           0           0
           0           0           0
Size of Array integers2 is 10
Array after initialization:
           0           0           0           0
           0           0           0           0
           0           0
Enter 17 integers:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

After input, the Arrays contain:
integers1:
           1           2           3           4
           5           6           7

integers2:
           8           9          10          11
          12          13          14          15
          16          17
Evaluating: integers1 != integers2
integers1 and integers2 are not equal

Size of Array integers3 is 7
Array after initialization:
           1           2           3           4
           5           6           7
Assigning integers2 to integers1:
integers1:
           8           9          10          11
          12          13          14          15
          16          17
integers2:
           8           9          10          11
```

```
          12          13          14          15
          16          17
Evaluating: integers1 == integers2
integers1 and integers2 are equal

integers1[5] is 13

Assigning 1000 to integers1[5]
integers1:
           8           9          10          11
          12        1000          14          15
          16          17
Attempt to assign 1000 to integers1[15]
Error: Subscript 15 out range
```

شکل ۸-۱۱ | برنامه تست کلاس Array.

ایجاد آرایه‌ها، نمایش سایز و محتویات آنها

برنامه با نمونه‌سازی دو شی از کلاس **Array** بنام‌های **integer1** (شکل ۸-۱۱، خط 12) با هفت عنصر، **integer2** (شکل ۸-۱۱، خط 13) با سایز پیش‌فرض **Array** یعنی 10 عنصر (مشخص شده توسط سازنده پیش‌فرض **Array** در شکل ۶-۱۱، خط 15) شروع می‌شود.

خطوط 18-16 از تابع عضو **getSize** برای تعیین سایز **integer1** استفاده کرده و محتویات **integer1** با استفاده از عملگر درج سربارگذاری شده **Array** در خروجی قرار داده می‌شود. خروجی نمونه این برنامه تایید می‌کند که عناصر **Array** بدرستی توسط سازنده با صفر مقداردهی اولیه شده‌اند. سپس خطوط 23-21 سایز آرایه **integer2** و محتویات آنرا توسط عملگر درج سربارگذاری شده، چاپ می‌کنند.

استفاده از عملگر درج سربارگذاری شده برای پر کردن آرایه

خط 26 به کاربر اعلان می‌کند تا 17 مقدار صحیح وارد سازد. خط 27 از عملگر استخراج سربارگذاری شده **Array** برای خواندن این مقادیر به هر دو آرایه استفاده کرده است. هفت مقدار اول در **integer1** و ده مقدار باقی مانده در **integer2** ذخیره می‌شوند. خطوط 31-29 دو آرایه را توسط عملگر درج سربارگذاری شده **Array** در خروجی قرار می‌دهند تا نشان دهند که عملیات ورودی بدرستی صورت گرفته است.

استفاده از عملگر نابرابری سربارگذاری شده

خط 36 مبادرت به تست عملگر نابرابری سربارگذاری شده با ارزیابی شرط

```
integers1 != integers2
```

می‌کند. خروجی برنامه نشان می‌دهد که آرایه‌ها به راستی برابر نیستند.

مقداردهی اولیه آرایه جدید با کپی از محتویات یک آرایه موجود

خط 41 مبادرت به نمونه‌سازی آرایه سومی بنام **integers3** کرده و آنرا با کپی از آرایه **integers1** مقداردهی اولیه می‌نماید. با اینکار سازنده کپی کننده آرایه فعال شده و عناصر آرایه **integers1** را به **integers3** کپی می‌نماید. بزودی در مورد جزئیات سازنده کپی‌کننده صحبت خواهیم کرد. دقت کنید که سازنده کپی‌کننده می‌تواند با نوشتن خط 41 بصورت زیر هم فعال شود:

```
Array integers3 = integers1;
```

نماد تساوی در عبارت فوق عملگر تخصیص نمی‌باشد. زمانیکه یک نماد تساوی در اعلان یک شی ظاهر می‌شود، مبادرت به فراخوانی سازنده برای آن شی می‌کند. در اینحالت فقط یک آرگومان می‌تواند به سازنده ارسال شود.

خطوط 45-43 سایز **integers3** و محتویات آنرا توسط عملگر درج سربارگذاری شده **Array** چاپ می‌کنند تا نشان دهند که عناصر آرایه بدرستی توسط سازنده کپی کننده مقداردهی شده است.

استفاده از عملگر تخصیص سربارگذاری شده

خط 49 مبادرت به تست عملگر تخصیص سربارگذاری شده (=) با تخصیص دادن **integers2** به **integers1** می‌کند. خطوط 52-51 هر دو آرایه را برای نشان دادن اینکه عملیات تخصیص با موفقیت صورت گرفته چاپ می‌کنند. دقت کنید که **integers1** در ابتدای کار هفت مقدار صحیح در خود نگهداری کرده بود و برای نگهداری ده عنصر **integers2** تغییر سایز داده است. همانطوری که مشاهده می‌کنید، عملگر تخصیص سربارگذاری شده این عملیات تغییر سایز را به روشی انجام می‌دهد که از دید کد سرویس گیرنده پنهان است.

استفاده از عملگر برابری سربارگذاری شده

خط 57 از عملگر برابری سربارگذاری شده (==) برای تایید اینکه آرایه‌های **integers1** و **integers2** به راستی پس از تخصیص با هم برابر هستند، استفاده کرده است.

استفاده از عملگر شاخص سربارگذاری شده

خط 61 از عملگر شاخص سربارگذاری شده برای اشاره یا مراجعه به **integers1[5]** استفاده کرده است که عنصری در محدودهٔ **integers1** است. نام شاخص بعنوان *rvalue* (مقدار سمت راست) برای چاپ مقدار ذخیره شده در **integers1[5]** بکار گرفته شده است. خط 65 از **integers1[5]** بعنوان یک *lvalue* (مقدار سمت چپ) تغییرپذیر در سمت چپ یک عبارت تخصیصی به منظور تخصیص یک مقدار جدید، 1000 به عنصر 5 از **integers1** استفاده کرده است. شاهد خواهید بود که **operator[]** یک مراجعه برای

استفاده بعنوان *lvalue* اصلاح‌پذیر پس از عملگر برگشت می‌دهد که نشان دهد که 5 یک شاخص معتبر برای **integers1** است.

خط 70 مبادرت به تخصیص مقدار 1000 به **integers1[15]** می‌کند، که عنصری خارج از محدودهٔ یا مرز آرایه می‌باشد. در این مثال، **operator[]** تعیین می‌کند که شاخص خارج از محدودهٔ بوده، یک پیغام چاپ کرده و برنامه خاتمه می‌پذیرد. دقت کنید که خط 70 در برنامه را متمایز کرده‌ایم تا بر این نکته تاکید کند که دسترسی به عنصری خارج از محدوده، خطا بدنبال خواهد داشت. این خطا از نوع خطای منطقی زمان اجرا بوده و یک خطای کامپایل نمی‌باشد.

جالب اینکه، عملگر شاخص آرایه **[]** فقط محدود به استفاده در آرایه‌ها نیست. برای مثال می‌توان از آن برای انتخاب عناصر از انواع کلاس‌های حامل نظیر لیست‌های پیوندی، رشته‌ها و واژه‌نامه استفاده کرد. همچنین پس از تعریف **operator[]**، شاخص دیگر مجبور نیست که حتماً یک مقدار صحیح باشد، کاراکتر، رشته، مقادیر اعشاری و حتی شی‌های تعریف شده توسط کاربر هم می‌توانند بکار گرفته شوند.

تعریف کلاس Array

اکنون که متوجه نحوه عملکرد برنامه شده‌اید، اجازه دهید به سراغ سرآیند کلاس برویم (شکل ۶-۱۱). همانطوری که به هر تابع عضو در سرآیند مراجعه می‌کنیم به توضیح پیاده‌سازی تابع در شکل ۷-۱۱ می‌پردازیم. در شکل ۶-۱۱، خطوط 36-35 عرضه کننده اعضای داده **private** کلاس **Array** هستند. هر شی **Array** متشکل از یک عضو **size** است که نشاندهنده تعداد عناصر در آرایه بوده و یک اشاره‌گر صحیح بنام **ptr** که به یک آرایه صحیح مبتنی بر اشاره‌گر و اخذ شده بفرم دینامیکی اشاره دارد، این آرایه توسط شی **Array** مدیریت می‌شود.

سربارگذاری عملگرهای درج و استخراج بعنوان friend

خطوط 13-12 از شکل ۶-۱۱ مبادرت به اعلان عملگرهای درج و استخراج سربارگذاری شده بعنوان دوستان (friend) کلاس **Array** کرده‌اند. زمانیکه کامپایلر به عبارتی مانند **cout<<arrayObject** برمی‌خورد، مبادرت به احضار تابع سراسری **operator**<< با فراخوانی

```
operator<<(cout, arrayObjet)
```

و زمانیکه کامپایلر به عبارتی مانند **cin>>arrayObject** برمی‌خورد، مبادرت به احضار تابع سراسری **operator**>> با فراخوانی

```
operator>>(cin, arrayObject)
```

می‌نماید. مجدداً توجه کنید که این توابع عملگر درج و استخراج نمی‌توانند عضو کلاس **Array** باشند، چرا که شی **Array** همیشه در طرف راست عملگر درج و استخراج جای داده می‌شود. اگر این توابع

عملگر اعضای از کلاس **Array** باشند، مجبور هستیم از عبارات غیراستادانه و ضعیف زیر برای چاپ و ورود آرایه استفاده کنیم:

```
arrayObject << cout;
arrayObject >> cin;
```

چنین عباراتی می‌توانند اکثر برنامه‌نویسان C++ را سردرگم کنند.

تابع **operator>>** (تعریف شده در شکل ۷-۱۱، خطوط 144-127) تعداد عناصر را براساس **size** از آرایه صحیح که **ptr** به آن اشاره می‌کند چاپ می‌نماید. تابع **operator<<** (تعریف شده در شکل ۷-۱۱، خطوط 124-118) بصورت مستقیم داده‌ها را به آرایه‌ای که **ptr** به آن اشاره دارد، وارد می‌سازد. هر یک از این توابع عملگر یک مراجعه برگشت می‌دهند تا بتوان عبارات خروجی یا ورودی پشت سرهم داشت.

دقت کنید که هر یک از این توابع دارای دسترسی به داده **private** آرایه هستند، چرا که این توابع بعنوان **friend** کلاس **Array** اعلام شده‌اند. همچنین توجه کنید که می‌توان توابع **getSize** و **operator[]** را توسط **operator>>** و **operator<<** بکار گرفت، در چنین حالتی نیازی نیست که این توابع عملگر، دوستان کلاس **Array** باشند. با این وجود، فراخوانی بیشتر تابع سبب افزایش زمان اجرا می‌شود.

سازنده پیش‌فرض Array

خط 15 از شکل ۶-۱۱ مبادرت به اعلان سازنده پیش‌فرض برای کلاس کرده و سایز اولیه آنرا با 10 عنصر مشخص ساخته است. زمانیکه کامپایلر اعلانی همانند خط 13 در شکل ۸-۱۱ را مشاهده می‌کند، مبادرت به احضار سازنده پیش‌فرض **Array** می‌کند. سازنده پیش‌فرض (تعریف شده در شکل ۷-۱۱، خطوط 18-25) شروع به ارزیابی و تخصیص آرگومان به عضو داده **size** کرده، از **new** برای بدست آوردن حافظه برای این آرایه مبتنی بر اشاره‌گر استفاده کرده و اشاره‌گر برگشتی توسط **new** را به عضو داده **ptr** تخصیص می‌دهد. سپس سازنده از یک عبارت **for** برای تنظیم تمام مقادیر آرایه با صفر استفاده کرده است.

سازنده کپی کننده Array

خط 16 از شکل ۶-۱۱ یک سازنده کپی کننده (تعریف شده در شکل ۷-۱۱، خطوط 36-29) اعلان کرده است که مبادرت به مقداردهی اولیه آرایه با تهیه کپی از روی یک شی آرایه موجود می‌کند. انجام چنین عملی (کپی) بایستی بدقت صورت گیرد تا از اشاره دادن، اشاره‌گر هر دو آرایه به یک مکان در حافظه جلوگیری شود. سازنده‌های کپی کننده زمانی بکار گرفته می‌شود که یک کپی از شی مورد نیاز باشد، همانند زمانیکه یک شی به روش مقدار به یک تابع ارسال می‌شود، یک شی به روش مقدار از یک تابع برگشت داده می‌شود یا مقداردهی اولیه یک شی با کپی از روی شی دیگر از همان کلاس. سازنده کپی

کننده زمانیکه یک شی از کلاس **Array** نمونه‌سازی و با شی دیگر از کلاس **Array** مقداردهی می‌شود، همانند اعلان موجود در خط 41 از شکل ۸-۱۱ فراخوانی می‌گردد.

سازنده کپی کننده **Array** از یک مقداردهی کننده (شکل ۷-۱۱، خط 30) برای کپی سایز مقداردهی کننده **Array** به عضو داده **size** استفاده کرده است. همچنین از **new** (خط 32) برای بدست آوردن حافظه برای این آرایه مبتنی بر اشاره‌گر و تخصیص اشاره‌گر برگشتی توسط **new** به عضو داده **ptr** استفاده کرده است. سپس سازنده کپی کننده با استفاده از یک عبارت **for** مبادرت به کپی تمام عناصر از آرایه مقداردهی کننده به آرایه جدید می‌کند.

نابود کننده Array

خط 17 از شکل ۶-۱۱ مبادرت به اعلان نابوده کننده برای کلاس کرده است (تعریف شده در شکل ۷-۱۱، خطوط 42-39). نابود کننده زمانی برای یک شی از کلاس **Array** احضار می‌شود که از قلمرو خارج شده باشد. نابود کننده از **[]delete** برای رهاسازی حافظه اخذ شده دینامیکی توسط **new** استفاده کرده است.

تابع عضو getSize

خط 18 از شکل ۶-۱۱ تابع **getSize** (تعریف شده در شکل ۷-۱۱، خطوط 48-45) را اعلان کرده است که تعداد عناصر در آرایه را برگشت می‌دهد.

عملگر تخصیص سربارگذاری شده

خط 20 از شکل ۶-۱۱ مبادرت به اعلان عملگر تخصیص سربارگذاری شده برای کلاس کرده است. زمانیکه کامپایلر به عبارت **integers1= integers2** در خط 49 از شکل ۸-۱۱ می‌رسد، تابع عضو **operator=** را با فراخوانی عبارت زیر احضار می‌کند.

```
integers1.operator=(integers2)
```

پیاده‌سازی تابع عضو **operator=** (شکل ۷-۱۱، خطوط 70-52) اقدام به تست خود تخصیصی (خط 54) می‌کند که در آن یک شی از کلاس **Array** به خودش تخصیص می‌یابد. زمانیکه **this** معادل با آدرس عملوند **right** باشد، پس مبادرت به خود تخصیصی شده است و از اینرو تخصیص به کنار گذاشته می‌شود (یعنی در حال حاضر شی خودش است). اگر نتیجه کار یک خود تخصیصی نباشد، پس تابع عضو تعیین می‌کند که آیا سایز دو آرایه با هم برابرند یا خیر (خط 58)، اگر برابر باشند، مقادیر آرایه اصلی در سمت چپ شی **Array** مجدداً اخذ نمی‌شود. در غیر اینصورت **operator=** از **delete** (خط 60) برای رها کردن حافظه اولیه اخذ شده برای آرایه هدف استفاده کرده، سایز آرایه منبع را به سایز (size) آرایه هدف

کپی می کند (خط 61)، از **new** برای اخذ حافظه برای آرایه هدف استفاده کرده و اشاره‌گر برگشتی از **new** را در **ptr** قرار می‌دهد. سپس عبارت **for** در خطوط 66-65 شروع به کپی عناصر آرایه از آرایه منبع به آرایه هدف می‌کند صرفنظر از اینکه خود تخصیصی رخ می‌دهد یا خیر، تابع عضو مبادرت به برگشت شی جاری (یعنی **this*** در خط 69) بعنوان یک مراجعه ثابت می‌کند، چنین کاری امکان تخصیص پشت سرهم همانند **x=y=z** را فراهم می‌آورد. اگر خود تخصیصی رخ دهد و تابع **operator=** اینحالت را تست نکند، **operator** مبادرت به حذف حافظه دینامیکی مرتبط با شی **Array** می‌کند، قبل از اینکه عملیات تخصیص کامل شود. در این وضعیت **ptr** به حافظه‌ای اشاره دارد که قبلاً بازپس گرفته شده است، و چنین کاری می‌تواند برنامه را بسوی *خطاهای زمان اجرای عظیم* (*fatal runtime error*) رهنمون سازد.

عملگرهای تساوی نابرابری سربارگذاری شده

خط 21 از شکل ۶-۱۱ مبادرت به اعلان عملگر تساوی سربارگذاری شده (==) برای کلاس کرده است. زمانیکه کامپایلر به عبارت **integers1== integers2** در خط 57 از شکل ۸-۱۱ می‌رسد، تابع عضو **operator==** را با فراخوانی عبارت زیر احضار می‌کند

```
integers1.operator==(integers2)
```

تابع عضو **operator==** (تعریف شده در شکل ۷-۱۱، خطوط 84-74) بلافاصله اگر مقدار **size** آرایه‌ها با هم برابر نباشند، **false** برگشت می‌دهد. در غیر اینصورت، **operator==** شروع به مقایسه هر جفت عناصر می‌کند. اگر همگی با هم برابر باشند، تابع مقدار **true** برگشت می‌دهد. با اولین برخورد به جفت عنصر غیر برابر، تابع بلافاصله مقدار **false** برگشت خواهد داد.

خطوط 27-24 از سرآیند فایل تعریف کننده عملگر نابرابری سربارگذاری شده (**!=**) برای کلاس هستند. تابع عضو **operator!=** از تابع **operator==** سربارگذاری شده برای تعیین اینکه یک آرایه با دیگری برابر است یا خیر استفاده می‌کند، سپس نتیجه مقتضی را برگشت می‌دهد. نوشتن **operator!=** به این روش به برنامه‌نویس امکان می‌دهد تا از **operator==** استفاده مجدد کند که نتیجه آن کاهش کدنویسی برای کلاس است. همچنین توجه کنید که کل تعریف تابع برای **operator!=** در فایل سرآیند **Array** قرار دارد. اینحالت به کامپایلر اجازه می‌دهد تا بصورت *inline* از **operator!=** استفاده کرده و جلوی فراخوانی اضافی تابع گرفته شود.

عملگر شاخص سربارگذاری شده

خطوط 30 و 33 از شکل ۶-۱۱ دو عملگر شاخص سربارگذاری شده اعلان کرده‌اند (تعریف شده در شکل ۷-۱۱ در خطوط 88-99 و 103-114). زمانیکه کامپایلر به عبارت **integers1[5]** (شکل ۸-۱۱، خط 61) می‌رسد، مبادرت به احضار تابع عضو سربارگذاری شده مقتضی **operator[]** با فراخوانی

```
integers1.operator[](5)
```

می‌کند. کامپایلر فراخوانی را بر روی نسخه ثابت **const** از **operator[]** انجام می‌دهد (شکل ۷-۱۱، خطوط 103-114)، زمانیکه عملگر شاخص بر روی یک شی آرایه ثابت بکار گرفته شده باشد. برای مثال، اگر شی ثابت **z** با عبارت زیر نمونه‌سازی شده باشد

```
const Array z(5);
```

پس نسخه ثابت از **operator[]** برای اجرای عبارتی مانند عبارت زیر لازم است

```
cout << z[3] << endl;
```

بخاطر داشته باشید که برنامه فقط می‌تواند توابع عضو ثابت **s** از یک شی ثابت را احضار کند.

هر تعریفی از **operator[]** تعیین می‌کند که آیا شاخص یک آرگومان در محدودهٔ دریافت می‌کند یا خیر. اگر چنین نباشد، هر تابع یک پیغام خطا چاپ کرده و برنامه با فراخوانی تابع **exit** خاتمه می‌پذیرد (سرآیند <**cstdlib**>). اگر شاخص در محدودهٔ قرار داشته باشد، نسخه غیرثابت **operator[]** عنصر آرایه مناسب را بعنوان یک مراجعه برگشت می‌دهد. از اینروست که می‌تواند بعنوان یک *lvalue* تغییرپذیر بکار گرفته شود (مثلاً در سمت چپ یک عبارت تخصیص). اگر شاخص در محدودهٔ قرار داشته باشد، نسخه ثابت **operator[]** یک کپی از عنصر مقتضی از آرایه را برگشت می‌دهد. کاراکتر برگشتی یک *rvalue* است.

۹-۱۱ تبدیل مابین نوع‌ها

اکثر برنامه‌ها مبادرت به پردازش اطلاعات از نوع‌های مختلف می‌کنند. گاهی اوقات تمام عملیات «در درون یک نوع» باقی می‌ماند. برای مثال، جمع یک **int** با یک **int**، یک **int** تولید می‌کند (مادامیکه نتیجه بدست آمده بسیار بزرگتر از **int** نباشد). با این همه، گاهی اوقات ضروری است که بتوان یک داده از یک نوع را به نوع دیگری تبدیل کرد. اینکار می‌تواند در هنگام تخصیص، در محاسبات، ارسال مقادیر به توابع و مقادیر برگشتی از توابع صورت گیرد. کامپایلر از نحوه تبدیلات مشخص در میان نوع‌های بنیادین مطلع است (همانطوری که در فصل ششم توضیح داده شد). اما تکلیف نوع‌های تعریف شده توسط کاربر چیست؟ کامپایلر با نحوه تبدیل مابین نوع‌های تعریف شده توسط کاربر و نوع‌های بنیادین مطلع نیست. از اینرو بایستی خود برنامه‌نویس نحوه انجام اینکار را مشخص نماید. چنین تبدیلاتی را می‌توان با *سازنده‌های تبدیل* انجام داد، سازنده‌های تک آرگومانی که شی‌ها از نوع‌های دیگر را (شامل نوع‌های بنیادین) به

شی‌های از یک کلاس خاص تبدیل می‌کنند. در بخش ۱۰-۱۱ از یک سازنده تبدیل برای تبدیل رشته‌های **char *** به شی‌های کلاس **String** استفاده کرده‌ایم.

عملگر تبدیل (که عملگر *cast* هم نامیده می‌شود) می‌تواند برای تبدیل یک شی از یک کلاس به یک شی از کلاس دیگر یا یک شی از نوع بنیادین بکار گرفته شود. چنین عملگر تبدیلی باید یک تابع عضو غیراستاتیک باشد. نمونه اولیه تابع

```
A::operator char *() const;
```

یک عملگر تبدیل سربارگذاری شده برای تبدیل یک شی از نوع تعریف شده توسط کاربر **A** به یک شی موقت ***char** اعلان کرده است. تابع اعلان شده **const** (ثابت) می‌باشد چرا که نمی‌تواند شی اصلی را دچار تغییر سازد. یک تابع عملگر تبدیل سربارگذاری شده نمی‌تواند نوع برگشتی را مشخص نماید، نوع برگشتی، نوعی است که شی به آن تبدیل خواهد شد. اگر **s** یک شی از کلاسی باشد، زمانیکه کامپایلر با عبارت **static_cast<char*>(s)** مواجه شود، عبارت زیر را فراخوانی می‌کند

```
s.operator char *()
```

عملوند **s** شی از کلاس **s** است که تابع عضو **operator char *** را احضار می‌کند.

می‌توان توابع عملگر تبدیل سربارگذاری شده را برای تبدیل شی‌ها از نوع تعریف شده توسط کاربر به نوع‌های بنیادین یا شی‌ها از شی دیگر تعریف کرد. نمونه اولیه

```
A::operator int() const;
A::operator OtherClass() const;
```

توابع عملگر تبدیل سربارگذاری شده را اعلان کرده که می‌تواند به ترتیب یک شی از نوع تعریف شده کاربر **A** را به یک نوع **int** یا یک شی از نوع تعریف شده توسط کاربر **OtherClass** تبدیل کند.

۱۰-۱۱ مبحث آموزشی: کلاس String

با هدف، داشتن یک تمرین مناسب از مبحث سربارگذاری، اقدام به ایجاد کلاس **String** متعلق بخود می‌کنیم که قادر به ایجاد و دستکاری رشته‌ها است (شکل‌های ۹-۱۱ الی ۱۱-۱۱). البته کتابخانه استاندارد C++ کلاس **string** مشابه و قدرتمندی دارد. از کلاس استاندارد **string** در بخش ۱۱-۱۳ در یک مثال استفاده می‌کنیم و در فصل هیجدهم به دقت به این کلاس می‌پردازیم. اما برای این لحظه، از ویژگی سربارگذاری عملگر به منظور ساخت کلاس **String** متعلق بخودمان استفاده می‌کنیم.

ابتدا، به معرفی فایل سرآیند کلاس **String** می‌پردازیم. در مورد داده خصوصی بکار رفته در عرضه شی‌های **String** توضیح می‌دهیم. سپس به سراغ واسط **public** کلاس رفته و هر یک از سرویس‌های کلاس را توضیح می‌دهیم. به بررسی تعاریف تابع عضو برای کلاس **String** می‌پردازیم. برای هر تابع

عملگر سربارگذاری شده، کدی را که سبب احضار تابع عملگر سربارگذاری شده است، عرضه کرده و توضیحی از نحوه عملکرد آنها خواهیم داد.

تعریف کلاس String

اکنون اجازه دهید به سراغ فایل سرآیند کلاس **String** در شکل ۹-۱۱ برویم. کار را با ارائه دهنده یک رشته داخلی مبتنی بر اشاره‌گر شروع می‌کنیم. خطوط 56-55 اعضای داده **private** را اعلان می‌کنند. کلاس **String** دارای یک فیلد **Length** می‌باشد که نشاندهنده تعداد کاراکترها در رشته است و شامل کاراکتر **null** در انتهای رشته نمی‌باشد و دارای یک اشاره‌گر بنام **sPtr** است که به حافظه اخذ شده دینامیکی برای رشته کاراکتری اشاره دارد.

```
1  // Fig. 11.9: String.h
2  // String class definition.
3  #ifndef STRING_H
4  #define STRING_H
5
6  #include <iostream>
7  using std::ostream;
8  using std::istream;
9
10 class String
11 {
12    friend ostream &operator<<( ostream &, const String & );
13    friend istream &operator>>( istream &, String & );
14 public:
15    String( const char * = "" ); // conversion/default constructor
16    String( const String & ); // copy constructor
17    ~String(); // destructor
18
19    const String &operator=( const String & ); // assignment operator
20    const String &operator+=( const String & ); // concatenation operator
21
22    bool operator!() const; // is String empty?
23    bool operator==( const String & ) const; // test s1 == s2
24    bool operator<( const String & ) const; // test s1 < s2
25
26    // test s1 != s2
27    bool operator!=( const String &right ) const
28    {
29       return !( *this == right );
30    } // end function operator!=
31
32    // test s1 > s2
33    bool operator>( const String &right ) const
34    {
35       return right < *this;
36    } // end function operator>
37
38    // test s1 <= s2
39    bool operator<=( const String &right ) const
40    {
41       return !( right < *this );
42    } // end function operator <=
43
44    // test s1 >= s2
45    bool operator>=( const String &right ) const
46    {
47       return !( *this < right );
48    } // end function operator>=
49
```

```
50    char &operator[]( int ); // subscript operator (modifiable lvalue)
51    char operator[]( int ) const; // subscript operator (rvalue)
52    String operator()( int, int = 0 ) const; // return a substring
53    int getLength() const; // return string length
54 private:
55    int length; // string length (not counting null terminator)
56    char *sPtr; // pointer to start of pointer-based string
57
58    void setString( const char * ); // utility function
59 }; // end class String
60
61 #endif
```

شکل ۹-۱۱ | تعریف کلاس String با سربارگذاری عملگر.

```
1   // Fig. 11.10: String.cpp
2   // Member-function definitions for class String.
3   #include <iostream>
4   using std::cerr;
5   using std::cout;
6   using std::endl;
7
8   #include <iomanip>
9   using std::setw;
10
11  #include <cstring> // strcpy and strcat prototypes
12  using std::strcmp;
13  using std::strcpy;
14  using std::strcat;
15
16  #include <cstdlib> // exit prototype
17  using std::exit;
18
19  #include "String.h" // String class definition
20
21  // conversion (and default) constructor converts char * to String
22  String::String( const char *s )
23     : length( ( s != 0 ) ? strlen( s ) : 0 )
24  {
25     cout << "Conversion (and default) constructor: " << s << endl;
26     setString( s ); // call utility function
27  } // end String conversion constructor
28
29  // copy constructor
30  String::String( const String &copy )
31     : length( copy.length )
32  {
33     cout << "Copy constructor: " << copy.sPtr << endl;
34     setString( copy.sPtr ); // call utility function
35  } // end String copy constructor
36
37  // Destructor
38  String::~String()
39  {
40     cout << "Destructor: " << sPtr << endl;
41     delete [] sPtr; // release pointer-based string memory
42  } // end ~String destructor
43
44  // overloaded = operator; avoids self assignment
45  const String &String::operator=( const String &right )
46  {
47     cout << "operator= called" << endl;
48
49     if ( &right != this ) // avoid self assignment
50     {
51        delete [] sPtr; // prevents memory leak
52        length = right.length; // new String length
53        setString( right.sPtr ); // call utility function
54     } // end if
55     else
56        cout << "Attempted assignment of a String to itself" << endl;
```

```
57
58    return *this; // enables cascaded assignments
59 } // end function operator=
60
61 // concatenate right operand to this object and store in this object
62 const String &String::operator+=( const String &right )
63 {
64    size_t newLength = length + right.length; // new length
65    char *tempPtr = new char[ newLength + 1 ]; // create memory
66
67    strcpy( tempPtr, sPtr ); // copy sPtr
68    strcpy( tempPtr + length, right.sPtr ); // copy right.sPtr
69
70    delete [] sPtr; // reclaim old space
71    sPtr = tempPtr; // assign new array to sPtr
72    length = newLength; // assign new length to length
73    return *this; // enables cascaded calls
74 } // end function operator+=
75
76 // is this String empty?
77 bool String::operator!() const
78 {
79    return length == 0;
80 } // end function operator!
81
82 // Is this String equal to right String?
83 bool String::operator==( const String &right ) const
84 {
85    return strcmp( sPtr, right.sPtr ) == 0;
86 } // end function operator==
87
88 // Is this String less than right String?
89 bool String::operator<( const String &right ) const
90 {
91    return strcmp( sPtr, right.sPtr ) < 0;
92 } // end function operator<
93
94 // return reference to character in String as a modifiable lvalue
95 char &String::operator[]( int subscript )
96 {
97    // test for subscript out of range
98    if ( subscript < 0 || subscript >= length )
99    {
100      cerr << "Error: Subscript " << subscript
101         << " out of range" << endl;
102      exit( 1 ); // terminate program
103   } // end if
104
105   return sPtr[ subscript ]; // non-const return; modifiable lvalue
106 } // end function operator[]
107
108 // return reference to character in String as rvalue
109 char String::operator[]( int subscript ) const
110 {
111    // test for subscript out of range
112    if ( subscript < 0 || subscript >= length )
113    {
114       cerr << "Error: Subscript " << subscript
115            << " out of range" << endl;
116       exit( 1 ); // terminate program
117    } // end if
118
119    return sPtr[ subscript ]; // returns copy of this element
120 } // end function operator[]
121
122 // return a substring beginning at index and of length subLength
123 String String::operator()( int index, int subLength ) const
124 {
125    // if index is out of range or substring length < 0,
126    // return an empty String object
```

```
127    if ( index < 0 || index >= length || subLength < 0 )
128       return ""; // converted to a String object automatically
129
130    // determine length of substring
131    int len;
132
133    if ( ( subLength == 0 ) || ( index + subLength > length ) )
134       len = length - index;
135    else
136       len = subLength;
137
138    // allocate temporary array for substring and
139    // terminating null character
140    char *tempPtr = new char[ len + 1 ];
141
142    // copy substring into char array and terminate string
143    strncpy( tempPtr, &sPtr[ index ], len );
144    tempPtr[ len ] = '\0';
145
146    // create temporary String object containing the substring
147    String tempString( tempPtr );
148    delete [] tempPtr; // delete temporary array
149    return tempString; // return copy of the temporary String
150 } // end function operator()
151
152 // return string length
153 int String::getLength() const
154 {
155    return length;
156 } // end function getLength
157
158 // utility function called by constructors and operator=
159 void String::setString( const char *string2 )
160 {
161    sPtr = new char[ length + 1 ]; // allocate memory
162
163    if ( string2 != 0 ) // if string2 is not null pointer, copy contents
164       strcpy( sPtr, string2 ); // copy literal to object
165    else // if string2 is a null pointer, make this an empty string
166       sPtr[ 0 ] = '\0'; // empty string
167 } // end function setString
168
169 // overloaded output operator
170 ostream &operator<<( ostream &output, const String &s )
171 {
172    output << s.sPtr;
173    return output; // enables cascading
174 } // end function operator<<
175
176 // overloaded input operator
177 istream &operator>>( istream &input, String &s )
178 {
179    char temp[ 100 ]; // buffer to store input
180    input >> setw( 100 ) >> temp;
181    s = temp; // use String class assignment operator
182    return input; // enables cascading
183 } // end function operator>>
```

شکل ۱۰-۱۱ | تعریف تابع عضو کلاس String و تابع friend.

```
1  // Fig. 11.11: fig11_11.cpp
2  // String class test program.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6  using std::boolalpha;
7
8  #include "String.h"
9
10 int main()
11 {
```

```
12    String s1( "happy" );
13    String s2( " birthday" );
14    String s3;
15
16    // test overloaded equality and relational operators
17    cout << "s1 is \"" << s1 << "\"; s2 is \"" << s2
18       << "\"; s3 is \"" << s3 << '\"'
19       << boolalpha << "\n\nThe results of comparing s2 and s1:"
20       << "\ns2 == s1 yields " << ( s2 == s1 )
21       << "\ns2 != s1 yields " << ( s2 != s1 )
22       << "\ns2 >  s1 yields " << ( s2 > s1 )
23       << "\ns2 <  s1 yields " << ( s2 < s1 )
24       << "\ns2 >= s1 yields " << ( s2 >= s1 )
25       << "\ns2 <= s1 yields " << ( s2 <= s1 );
26
27
28    // test overloaded String empty (!) operator
29    cout << "\n\nTesting !s3:" << endl;
30
31    if ( !s3 )
32    {
33       cout << "s3 is empty; assigning s1 to s3;" << endl;
34       s3 = s1; // test overloaded assignment
35       cout << "s3 is \"" << s3 << "\"";
36    } // end if
37
38    // test overloaded String concatenation operator
39    cout << "\n\ns1 += s2 yields s1 = ";
40    s1 += s2; // test overloaded concatenation
41    cout << s1;
42
43    // test conversion constructor
44    cout << "\n\ns1 += \" to you\" yields" << endl;
45    s1 += " to you"; // test conversion constructor
46    cout << "s1 = " << s1 << "\n\n";
47
48    // test overloaded function call operator () for substring
49    cout << "The substring of s1 starting at\n"
50       << "location 0 for 14 characters, s1(0, 14), is:\n"
51       << s1( 0, 14 ) << "\n\n";
52
53    // test substring "to-end-of-String" option
54    cout << "The substring of s1 starting at\n"
55       << "location 15, s1(15), is: "
56       << s1( 15 ) << "\n\n";
57
58    // test copy constructor
59    String *s4Ptr = new String( s1 );
60    cout << "\n*s4Ptr = " << *s4Ptr << "\n\n";
61
62    // test assignment (=) operator with self-assignment
63    cout << "assigning *s4Ptr to *s4Ptr" << endl;
64    *s4Ptr = *s4Ptr; // test overloaded assignment
65    cout << "*s4Ptr = " << *s4Ptr << endl;
66
67    // test destructor
68    delete s4Ptr;
69
70    // test using subscript operator to create a modifiable lvalue
71    s1[ 0 ] = 'H';
72    s1[ 6 ] = 'B';
73    cout << "\ns1 after s1[0] = 'H' and s1[6] = 'B' is: "
74       << s1 << "\n\n";
75
76    // test subscript out of range
77    cout << "Attempt to assign 'd' to s1[30] yields:" << endl;
78    s1[ 30 ] = 'd'; // ERROR: subscript out of range
79    return 0;
80 } // end main
```

```
Conversion (and default) constructor: happy
```

```
Conversion (and default) constructor: birthday
Conversion (and default) constructor:
s1 is "happy"; s2 is " birthday"; s3 is ""

The results of comparing s2 and s1:
s2 == s1 yields false
s2 != s1 yields true
s2 >  s1 yields false
s2 <  s1 yields true
s2 >= s1 yields false
s2 <= s1 yields true

Testing !s3:
s3 is empty; assiging s1 to s3;
operator= called
s3 is "happy"

s1 += s2 yields s1 = happy birthday

s1 += " to you" yields
Conversion (and default) constructor: to you
Destructor: to you
s1 = happy birthday to you

Conversion (and default) constractor: happy birthday
copy constructor: happy birthday
Destructor: happy birthday
The substring of s1 starting at
location 0 for 14 characters, s1(0, 14), is:
happy birthday

Destructor: happy birthday
Conversion (and default) constructor: to you
Copy constructor: to you
Destructor: to you
The substring of s1 starting at
location 15, s1(15), is: to you

Destructor: to you
Copy constructor: happy birthday to you

*s4Ptr = happy birthday to you

assigning *s4Ptr to *s4Ptr
operator= called
Attempted assignment of a String to itself

*s4Ptr = happy birthday to you
Destructor: happy birthday to you

s1 after s1[0] = 'H'and s1[6] = 'B'is: Happy Birthday to you

Attempt to assign 'd'to s1[30] yields:
Error: Subscript 30 out of range
```

شکل ۱۱-۱۱ | برنامه تست کننده کلاس String.

سربارگذاری عملگرهای درج و استخراج بعنوان friend

خطوط 12-13 (شکل ۹-۱۱) مبادرت به اعلان تابع عملگر درج سربارگذاری شده **operator**<< (تعریف شده شکل ۱۰-۱۱، خطوط 170-174) و تابع عملگر استخراج سربارگذاری شده **operator**>> (تعریف شده در شکل ۱۰-۱۱، خطوط 177-183) بعنوان **friend** کلاس کرده‌اند. پیاده‌سازی **operator**<< سر راست است. دقت کنید که **operator**<< محدود به تعداد کاراکترهای است که می‌توان به آرایه از **temp**

تا 99 با **setw** (خط 180) خواند، مکان صدم برای کاراکتر خاتمه دهنده ***null***، در نظر گرفته شده است. همچنین به نحوه استفاده از **operator=** (خط 181) در تخصیص رشته **temp** به شی **String** که **s** به آن اشاره دارد، توجه کنید. این عبارت سازنده تبدیل کننده را برای ایجاد کد شی **String** موقت که حاوی رشته‌ای به سبک C است احضار کرده، سپس رشته موقت **String** به **s** تخصیص می‌یابد. می‌توانیم ایجاد موقت شی **String** بکار رفته در اینجا را به کمک یک عملگر تخصیص سربارگذاری شده که پارامتری از نوع ***const char** دریافت می‌کند، برطرف نمائیم.

سازنده تبدیل کننده String

در خط 15 (شکل ۱۹-۱۱) یک سازنده تبدیل کننده اعلان شده است. این سازنده (تعریف شده در شکل ۱۰-۱۱، خطوط 27-22) یک آرگومان ***const char** دریافت (پیش‌فرض رشته تهی، شکل ۹-۱۱، خط 15) و شی **String** را مقداردهی اولیه می‌کند. هر سازنده تک آرگومانی را می‌توان بعنوان یک سازنده تبدیل کننده بکار گرفت. همانطوری که خواهید دید، چنین تبدیل کننده‌های به هنگام کار با رشته‌ای با آرگومان ***char** سودمند هستند. سازنده تبدیل کننده قادر به تبدیل یک رشته ***char** به یک شی **String** است، که سپس می‌تواند به شی **String** هدف تخصیص داده شود. مزیت این سازنده تبدیل کننده در این است که نیازی به تدارک دیدن یک عملگر تخصیص سربارگذاری شده برای تخصیص رشته‌های کاراکتری به شی‌های **String** ندارد. کامپایلر، سازنده تبدیل کننده را برای ایجاد یک شی **String** موقت که حاوی رشته کاراکتری است احضار کرده، سپس عملگر تخصیص سربارگذاری شده را برای انتساب شی **String** موقت به شی **String** دیگر احضار می‌کند.

سازنده تبدیل کننده **String** می‌تواند در اعلانی نظیر **s1("happy")** احضار شود. سازنده تبدیل کننده طول کاراکترهای موجود در آرگومان خود را محاسبه و آنرا به عضو داده **length** در لیست مقداردهی اولیه تخصیص می‌دهد. سپس خط 26 تابع کمکی **setString** (تعریف شده در شکل ۱۰-۱۱، خطوط 167-159) را که از **new** برای اخذ حافظه کافی برای عضو داده خصوصی **sptr** استفاده می‌کند فراخوانی کرده و از **strcpy** برای کپی رشته کاراکتری به حافظه‌ای که **sPtr** به آن اشاره دارد، استفاده می‌کند.

سازنده کپی کننده String

خط 16 در شکل ۹-۱۱ یک سازنده کپی کننده (تعریف شده در شکل ۱۰-۱۱، خطوط 35-30) اعلان کرده است که مبادرت به مقداردهی یک شی **String** با ایجاد یک کپی از روی یک شی **String** موجود می‌کند. همانند کلاس **Array** (شکل‌های ۶-۱۱ و ۷-۱۱)، انجام چنین عملی باید با دقت صورت گیرد تا هر دو شی **String** به یک حافظه اشاره نکند. عملگر سازنده کپی کننده همانند سازنده تبدیل کننده است،

بجز اینکه این سازنده فقط عضو **Length** از شی **String** منبع را به شی **String** هدف کپی می‌کند. دقت کنید که سازنده کپی کننده **setString** را برای ایجاد یک فضای جدید برای شی هدف فراخوانی می‌کند. اگر این سازنده فقط مبادرت به کپی **sPtr** در شی منبع به **sPtr** شی هدف می‌کرد، آنگاه هر دو شی به یک حافظه اشاره می‌کردند. با اجرای اولین نابود کننده، حافظه حذف می‌گردید و **sPtr** شی‌های دیگر بلاتکلیف باقی می‌ماندند (یعنی **sPtr** تبدیل به یک اشاره‌گر *dangling* یا آویزان می‌شود) و در نتیجه، خطاهای زمان اجرای بسیاری جدی رخ می‌دهد.

نابود کننده String

خط 17 از شکل ۹-۱۱ مبادرت به اعلان نابود کننده **String** (تعریف شده در شکل ۱۰-۱۱، خطوط 38-42) کرده است. نابود کننده با استفاده **[]delete** حافظه دینامیکی که **sPtr** به آن اشاره می‌کند، آزاد می‌سازد.

عملگر تخصیص سربارگذاری شده

خط 19 (شکل ۹-۱۱) تابع عملگر تخصیص سربارگذاری شده **operator=** را اعلان کرده است (تعریف شده در خطوط 45-59 شکل ۱۰-۱۱). زمانیکه کامپایلر به عبارتی مانند **string1=string2** می‌رسد، تابع زیر را فراخوانی می‌کند

```
string1.operator=(string2);
```

تابع عملگر تخصیص سربارگذاری شده **operator=** مبادرت به تست خود تخصیصی می‌کند. اگر این تخصیص، یک خود تخصیصی باشد، تابع نیازی به تغییر شی ندارد. اگر این تست حذف شود یا انجام نشود، تابع بلافاصله اقدام به حذف فضای شی هدف کرده و از اینرو رشته کاراکتری از دست می‌رود. در چنین حالتی دیگر اشاره‌گر به داده معتبر اشاره ندارد (نمونه کلاسیک این حالت اشاره‌گر *dangling* است). اگر نتیجه تست یک خود تخصیصی نباشد، تابع مبادرت به حذف حافظه کرده و فیلد **length** از شی منبع را به شی هدف کپی می‌کند. سپس **operator=** مبادرت به فراخوانی **setString** برای ایجاد یک فضای جدید برای شی هدف کرده و رشته کاراکتری را از شی منبع به شی هدف کپی می‌کند. خواه خود تخصیصی صورت گرفته باشد یا نباشد، **operator =** مبادرت به برگشت دادن **this** * می‌کند تا بتوان پشت سر هم تخصیص انجام داد.

عملگر جمع تخصیص سربارگذاری شده

در خط 20 از شکل ۹-۱۱ عملگر اتصال رشته سربارگذاری شده =+ اعلان شده است (تعریف شده در شکل ۱۰-۱۱، خطوط 74-62). زمانیکه کامپایلر به عبارتی نظیر **s1 += s2** می‌رسد (خط 40 از شکل ۱۱-۱۱)، کامپایلر تابع عضو را فراخوانی می‌کند

```
s1.operator+=(s2)
```

تابع **operator+=** مبادرت به محاسبه طول ترکیب شده از رشته متصل شده کرده و آنرا در متغیر محلی **newLength** ذخیره می‌کند، سپس یک اشاره‌گر موقت (**tempPtr**) ایجاد کرده و یک آرایه کاراکتری جدید اخذ می‌کند که بتوان رشته متصل شده را در آن ذخیره کرد.

سپس، **operator+=** از **strcpy** برای کپی کردن رشت کاراکتری اصلی از **sPtr** و **right.sPtr** به حافظه‌ای که **tempPtr** به آن اشاره دارد، استفاده می‌کند. دقت کنید مکانی که **strcpy** مبادرت به کپی اولین کاراکتر از **right.sPtr** می‌کند با محاسبه ریاضی اشاره‌گر یعنی **tempPtr + length** مشخص می‌شود. این محاسبه بر این نکته دلالت دارد که اولین کاراکتر از **rightsPtr** بایستی در موقعیت **length** در آرایه‌ای که **tempPtr** به آن اشاره دارد، قرار داده شود. سپس **operator+=** از **[]delete** برای رهاسازی فضای اشغال شده توسط رشته کاراکتری شی اصلی استفاده کرده، **tempPtr** را به **sPtr** تخصیص می‌دهد. از اینرو این شی **String** به رشته کاراکتری جدید اشاره می‌کند، **newLength** را به **length** تخصیص و از اینروست که این شی **String** حاوی طول رشته جدید بوده و **this *** را بعنوان یک **const String &** برگشت می‌دهد تا بتوان عملگرهای =+ پشت سرهم یا دنباله‌دار داشت.

عملگر نفی سربارگذاری شده

خط 22 از شکل ۹-۱۱ عملگر نفی سربارگذاری شده را اعلان کرده است (تعریف شده در شکل ۱۰-۱۱، خطوط 80-77). این عملگر تعیین می‌کند که آیا یک شی از کلاس **String** تهی است یا خیر. برای مثال، زمانیکه کامپایلر به عبارتی مانند **!string1** می‌رسد. فراخوانی تابع زیر را انجام می‌هد

```
string1.operator!()
```

این تابع فقط نتیجه تست را باز می‌گرداند خواه **length** برابر صفر باشد یا نباشد.

عملگرهای برابری و رابطه‌ای سربارگذاری شده

خطوط 24-23 از شکل ۹-۱۱ مبادرت به اعلان عملگر برابری سربارگذاری شده (تعریف شده در شکل ۱۰-۱۱، خطوط 86-83) و عملگر کوچکتر از، سربارگذاری شده (تعریف شده در شکل ۱۰-۱۱، خطوط 92-89) برای کلاس **String** کرده‌اند. این دو شبیه هم هستند، از اینرو اجازه دهید فقط به بررسی یک مثال

و آن هم برای عملگر == سربارگذاری شده بپردازیم. زمانیکه کامپایلر به عبارتی همانند **string1==string2** می‌رسد، تابع عضو زیر را فراخوانی می‌کند

```
string1.operator==(string2)
```

در صورتیکه **string1** معادل (برابر) با **string2** باشد، **true** برگشت می‌دهد. هر یک از عملگرها از تابع **strcmp** (از **<cstring>**) برای مقایسه رشته‌های کاراکتری در شی‌های **String** استفاده می‌کنند. تعدادی از برنامه‌نویسان C++ طرفدار استفاده از برخی توابع عملگر سربارگذاری شده برای پیاده‌سازی موارد دیگر هستند. از اینرو، عملگرهای **=!**، **<=** و **=>** (شکل ۹-۱۱، خطوط 48-27) برحسب **operator==** و **operator<** پیاده‌سازی شده‌اند. برای مثال، تابع سربارگذاری شده **operator>=** (پیاده‌سازی شده در خطوط 48-45 در فایل سرآیند) از عملگر سربارگذاری شده > برای تعیین اینکه یک شی **String** بزرگ‌تر یا برابر شی **String** دیگری است یا خیر، استفاده کرده است. دقت کنید که توابع عملگر برای **=!**، **<=** و **=>** در فایل سرآیند تعریف شده‌اند.

عملگرهای شاخص سربارگذاری شده

خطوط 51-50 در فایل سرآیند دو عملگر شاخص سربارگذاری شده اعلان کرده است (تعریف شده در شکل ۱۰-۱۱، خطوط 106-95 و 120-109) یکی برای رشته‌های غیرثابت و یکی برای رشته‌های ثابت. زمانیکه کامپایلر به عبارتی همانند **string1[0]** می‌رسد، تابع عضو زیر را فراخوانی می‌کند

```
string1.operator[](0)
```

هر پیاده‌سازی **operator[]** ابتدا مبادرت به ارزیابی شاخص می‌کند تا مطمئن گردد در محدوده صحیح قرار دارد. اگر شاخص در محدودهٔ قرار نداشته باشد، هر تابع یک پیغام خطا چاپ کرده و برنامه با فراخوانی **exit** خاتمه می‌پذیرد. اگر شاخص در محدودهٔ صحیح قرار داشته باشد، نسخه غیر ثابت **operator[]** یک **char &** به کاراکتر مقتضی از شی **String** برگشت می‌دهد، این **char &** می‌تواند بعنوان یک *lvalue* برای تغییر در یک کاراکتر خاص از شی **String** بکار گرفته شود. نسخه ثابت **operator[]** کاراکتر مقتضی از شی **String** برگشت می‌دهد که از آن می‌توان فقط بعنوان یک *rvalue* به منظور خواندن کاراکتر استفاده کرد.

عملگر فراخوانی تابع سربارگذاری شده

خط 52 از شکل ۹-۱۱ عملگر فراخوانی تابع سربارگذاری شده (تعریف شده در شکل ۱۰-۱۱، خطوط 150-123) را اعلان کرده است. این عملگر را برای انتخاب یک زیر رشته از یک **String** سربارگذاری کرده‌ایم. دو پارامتر صحیح مشخص کننده موقعیت شروع و طول زیر رشته‌ای است که از **String** انتخاب خواهد شد. اگر موقعیت یا مکان شروع در خارج از محدودهٔ قرار داشته باشد یا طول زیر رشته منفی باشد،

عملگر فقط یک رشته تهی برگشت خواهد داد. اگر طول زیر رشته صفر باشد، سپس زیر رشته تا انتهای شی رشته انتخاب می‌شود. برای مثال فرض کنید **string1** یک شی از **String** است و حاوی رشته "AEIOU" باشد. در عبارتی مانند **string1(2,2)** کامپایلر فراخوانی تابع عضو زیر را انجام می‌دهد

```
string1.operator()(2,2)
```

با اجرای این فراخوانی، یک شی **String** حاوی رشته "IO" تولید و یک کپی از آن شی برگشت داده می‌شود.

سربارگذاری عملگر فراخوانی تابع **()** ابزار قدرتمندی است چراکه توابع می‌توانند به تعداد دلخواه و ترکیبی پارامتر دریافت کنند. از اینرو می‌توانیم از این قابلیت در زمینه‌های مختلفی استفاده کنیم. یکی از موارد استفاده از عملگر فراخوانی تابع تغییر در نماد شاخص آرایه است. بجای استفاده از نماد براکت در آرایه‌های دوبعدی همانند **a[b][c]**، برخی از برنامه‌نویسان ترجیح می‌دهند با سربارگذاری عملگر فراخوانی تابع از نماد **a(b,c)** استفاده کنند. عملگر فراخوانی تابع سربارگذاری شده باید یک تابع عضو غیراستاتیک باشد. این عملگر فقط زمانی بکار گرفته می‌شود که «نام تابع» یک شی از کلاس **String** باشد.

تابع عضو getLength

خط 53 در شکل ۹-۱۱ تابعی بنام **getLength** (تعریف شده در شکل ۱۰-۱۱، خطوط 156-153) اعلان کرده است، که طول یک رشته را برگشت می‌دهد.

۱۱-۱۱ سربارگذاری ++ و --

نسخه‌های پیشوند و پسوند عملگرهای افزایشی و کاهشی را می‌توان سربارگذاری کرد. خواهیم دید که چگونه کامپایلر قادر به تشخیص نسخه پیشوند و نسخه پسوند از یک عملگر افزایش دهنده یا کاهش دهنده است.

برای سربارگذاری کردن عملگر افزایش دهنده به نحوی که امکان استفاده از هر دو نوع افزایش پیشوندی و پسوندی در آن وجود داشته باشد، بایستی هر عملگر سربارگذاری شده دارای یک امضاء یا هویت متمایز باشد، از اینروست که کامپایلر قادر به تعیین نسخه مورد نظر ++ خواهد بود. نسخه‌های پیشوندی دقیقاً همانند سایر عملگرهای پیشوندی غیربااینری سربارگذاری می‌شوند.

سربارگذاری عملگر پیشوند افزایشی

برای مثال، فرض کنید که می‌خواهیم 1 را به شی **d1** از کلاس **Date** اضافه کنیم. زمانیکه کامپایلر به عبارت پیش افزایشی **d1++** برخورد می کند، فراخوانی تابع زیر را انجام می‌دهد

```
d1.operator++()
```

نمونه اولیه این تابع عملگر بصورت زیر است

```
Date &operator++();
```

اگر عملگر افزایش دهنده پیشوندی بصورت یک تابع سراسری پیاده‌سازی شده باشد، پس زمانیکه کامپایلر به عبارت **d1**++ برسد، تابع زیر را فراخوانی خواهد کرد

```
operator++(d1)
```

نمونه اولیه این تابع عملگر می‌تواند بصورت زیر در کلاس **Date** اعلان شده باشد

```
Date &operator++(Date &);
```

سربارگذاری عملگر پسوند افزایشی

سربارگذاری عملگر پسوند افزایشی کمی کار دارد چرا که کامپایلر باید قادر به تشخیص و تمایز قائل شدن مابین امضاءهای توابع عملگر پیشوند و پسوند افزایشی باشد. قراردادی که در C++ پذیرفته شده این است که، زمانیکه کامپایلر به عبارت پس افزایشی **d1**++ می‌رسد، فراخوانی تابع عضو

```
d1.operator++(0)
```

را انجام می‌دهد. نمونه اولیه این تابع

```
Date operator++(int)
```

است. آرگومان صفر کاملاً یک «مقدار ساختگی» است که به کامپایلر امکان می‌دهد تا مابین توابع عملگر افزایش پیشوند و پسوند تمایز قائل شود. اگر افزایش پسوندی بصورت یک تابع سراسری پیاده‌سازی شده باشد، سپس به هنگام مشاهده عبارت **d1**++ توسط کامپایلر، فراخوانی زیر صورت می‌گیرد

```
operator++(d1, 0)
```

نمونه اولیه این تابع بصورت زیر است

```
Date operator++(Date &, int);
```

مجدداً، آرگومان صفر توسط کامپایلر برای تشخیص مابین عملگرهای افزایشی پسوندی و پیشوندی بعنوان توابع سراسری استفاده می‌شود. توجه کنید که عملگر افزایشی پسوندی شی‌های **Date** را به روش مقدار برگشت می‌دهد، در حالیکه عملگر افزایشی پیشوندی شی‌های **Date** را به روش مراجعه برگشت می‌دهد، چرا که عملگر افزایشی پسوندی یک شی موقت برگشت می‌دهد که حاوی مقدار اصلی از شی است، قبل از اینکه عملیات افزایش رخ داده باشد. C++ با چنین شی‌های بعنوان *rvalue* رفتار می‌کند که نمی‌توان از آنها در سمت چپ یک عبارت تخصیصی استفاده کرد. عملگر افزایشی پیشوندی، در واقع شی افزایش یافته را به همراه مقدار جدید برگشت می‌دهد. از چنین شی می‌توان بعنوان یک *lvalue* در یک عبارت شمارشی استفاده کرد.

هر عملی که بر روی سربارگذاری عملگرهای افزایشی پسوندی و پیشوندی در این بخش توضیح داده شد در ارتباط با سربارگذاری عملگرهای کاهش پسوندی و پیشوندی نیز کاربرد دارد. در بخش بعد از کلاس **Date** به همراه عملگرهای سربارگذاری شده ++ و -- استفاده می‌کنیم.

۱۲-۱۱ مبحث آموزشی: کلاس Date

برنامه موجود در شکل‌های ۱۲-۱۱ الی ۱۴-۱۱ به توصیف کاربردی از کلاس **Date** می‌پردازد. این کلاس از عملگرهای سربارگذاری شده پیش و پس افزایشی برای افزودن 1 به روز در یک شی **Date** استفاده می‌کند، و در صورتیکه نیاز باشد افزایش ماه و سال هم صورت می‌گیرد. فایل سرآیند **Date** (شکل ۱۲-۱۱) مشخص کرده که واسط سراسری **Date** شامل یک عملگر درج سربارگذاری شده (خط 11)، یک سازنده پیش‌فرض (خط 13)، تابع **setDate** (خط 14)، عملگر سربارگذاری شده پیش‌افزایشی (خط 15)، عملگر سربارگذاری شده پس‌افزایشی (خط 16)، عملگر سربارگذاری شده جمع تخصیصی =+ در خط 17، تابعی برای تست سالهای کبیسه (خط 18) و تابعی برای تعیین اینکه روز بدست آمده آخرین روز از ماه است یا خیر (خط 19) می‌باشد.

```
1  // Fig. 11.12: Date.h
2  // Date class definition.
3  #ifndef DATE_H
4  #define DATE_H
5
6  #include <iostream>
7  using std::ostream;
8
9  class Date
10 {
11    friend ostream &operator<<( ostream &, const Date & );
12 public:
13    Date( int m = 1, int d = 1, int y = 1900 ); // default constructor
14    void setDate( int, int, int ); // set month, day, year
15    Date &operator++(); // prefix increment operator
16    Date operator++( int ); // postfix increment operator
17    const Date &operator+=( int ); // add days, modify object
18    bool leapYear( int ) const; // is date in a leap year?
19    bool endOfMonth( int ) const; // is date at the end of month?
20 private:
21    int month;
22    int day;
23    int year;
24
25    static const int days[]; // array of days per month
26    void helpIncrement(); // utility function for incrementing date
27 }; // end class Date
28
29 #endif
```

شکل ۱۲-۱۱ | تعریف کلاس Date به همراه عملگرهای افزایشی سربارگذاری شده.

```
1  // Fig. 11.13: Date.cpp
2  // Date class member-function definitions.
3  #include <iostream>
4  #include "Date.h"
5
6  // initialize static member at file scope; one classwide copy
7  const int Date::days[] =
8     { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
9
10 // Date constructor
11 Date::Date( int m, int d, int y )
12 {
13    setDate( m, d, y );
14 } // end Date constructor
15
16 // set month, day and year
```

```
17 void Date::setDate( int mm, int dd, int yy )
18 {
19    month = ( mm >= 1 && mm <= 12 ) ? mm : 1;
20    year = ( yy >= 1900 && yy <= 2100 ) ? yy : 1900;
21
22    // test for a leap year
23    if ( month == 2 && leapYear( year ) )
24       day = ( dd >= 1 && dd <= 29 ) ? dd : 1;
25    else
26       day = ( dd >= 1 && dd <= days[ month ] ) ? dd : 1;
27 } // end function setDate
28
29 // overloaded prefix increment operator
30 Date &Date::operator++()
31 {
32    helpIncrement(); // increment date
33    return *this; // reference return to create an lvalue
34 } // end function operator++
35
36 // overloaded postfix increment operator; note that the
37 // dummy integer parameter does not have a parameter name
38 Date Date::operator++( int )
39 {
40    Date temp = *this; // hold current state of object
41    helpIncrement();
42
43    // return unincremented, saved, temporary object
44    return temp; // value return; not a reference return
45 } // end function operator++
46
47 // add specified number of days to date
48 const Date &Date::operator+=( int additionalDays )
49 {
50    for ( int i = 0; i < additionalDays; i++ )
51       helpIncrement();
52
53    return *this; // enables cascading
54 } // end function operator+=
55
56 // if the year is a leap year, return true; otherwise, return false
57 bool Date::leapYear( int testYear ) const
58 {
59    if ( testYear % 400 == 0 ||
60       ( testYear % 100 != 0 && testYear % 4 == 0 ) )
61       return true; // a leap year
62    else
63       return false; // not a leap year
64 } // end function leapYear
65
66 // determine whether the day is the last day of the month
67 bool Date::endOfMonth( int testDay ) const
68 {
69    if ( month == 2 && leapYear( year ) )
70       return testDay == 29; // last day of Feb. in leap year
71    else
72       return testDay == days[ month ];
73 } // end function endOfMonth
74
75 // function to help increment the date
76 void Date::helpIncrement()
77 {
78    // day is not end of month
79    if ( !endOfMonth( day ) )
80       day++; // increment day
81    else
82       if ( month < 12 ) // day is end of month and month < 12
83       {
84          month++; // increment month
85          day = 1; // first day of new month
86       } // end if
```

```
87        else // last day of year
88        {
89           year++; // increment year
90           month = 1; // first month of new year
91           day = 1; // first day of new month
92        } // end else
93  } // end function helpIncrement
94
95  // overloaded output operator
96  ostream &operator<<( ostream &output, const Date &d )
97  {
98     static char *monthName[ 13 ] = { "", "January", "February",
99        "March", "April", "May", "June", "July", "August",
100       "September", "October", "November", "December" };
101    output << monthName[ d.month ] << ' ' << d.day << ", " << d.year;
102     return output; // enables cascading
103  } // end function operator<<
```

شکل ۱۳-۱۱ | تعاریف تابع عضو و friend کلاس Date.

```
1  // Fig. 11.14: fig11_14.cpp
2  // Date class test program.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Date.h" // Date class definition
8
9  int main()
10 {
11    Date d1; // defaults to January 1, 1900
12    Date d2( 12, 27, 1992 ); // December 27, 1992
13    Date d3( 0, 99, 8045 ); // invalid date
14
15    cout << "d1 is " << d1 << "\nd2 is " << d2 << "\nd3 is " << d3;
16    cout << "\n\nd2 += 7 is " << ( d2 += 7 );
17
18    d3.setDate( 2, 28, 1992 );
19    cout << "\n\n  d3 is " << d3;
20    cout << "\n++d3 is " << ++d3 << " (leap year allows 29th)";
21
22    Date d4( 7, 13, 2002 );
23
24    cout << "\n\nTesting the prefix increment operator:\n"
25       << "  d4 is " << d4 << endl;
26    cout << "++d4 is " << ++d4 << endl;
27    cout << "  d4 is " << d4;
28
29    cout << "\n\nTesting the postfix increment operator:\n"
30       << "  d4 is " << d4 << endl;
31    cout << "d4++ is " << d4++ << endl;
32    cout << "  d4 is " << d4 << endl;
33    return 0;
34 } // end main
```

```
d1 is January 1, 1900
d2 is December 27, 1992
d3 is January 1, 1900

d2 += 7 is January 3, 1993

  d3 is February 28, 1992
++d3 is February 29, 1992 (leap year allows 29th)

Testing the prefix increment operator:
  d4 is July 13, 2002
++d4 is July 14, 2002
  d4 is July 14, 2002

Testing the postfix increment operator:
  d4 is July 14, 2002
d4++ is July 14, 2002
```

```
d4 is July 15, 2002
```

شکل ١٤-١١ | برنامه تست کلاس Date.

تابع **main** (شکل ۱۴-۱۱) سه شی **Date** ایجاد کرده است (خطوط 13-11)، **d1** بطور پیش‌فرض با January 1,1900، **d2** با December 27,1992 و **d3** با یک تاریخ غیر معتبر مقداردهی اولیه شده‌اند. سازنده **Date** (تعریف شده در شکل ۱۳-۱۱، خطوط 14-11) تابع **setDate** را به منظور ارزیابی ماه، روز و سال مشخص شده فراخوانی می‌کند. ماه غیر معتبر با 1، سال غیر معتبر با 1900 و روز غیر معتبر با 1 تنظیم می‌شود.

خطوط 16-15 از **main** هر یک از شی‌های ساخته شده از **Date** را با استفاده از عملگر درج سربارگذاری شده (تعریف شده در شکل ۱۳-۱۱، خطوط 103-96) در خروجی قرار می‌دهند. خط 16 از **main** از عملگر سربارگذاری شده =+ به منظور افزودن هفت روز به **d2** استفاده کرده است. خط 18 از تابع **setDate** برای تنظیم **d3** با February 28,1992 استفاده کرده که یک سال کبیسه است. سپس خط 20 بصورت پیش افزایشی **d3** را برای نمایش آن که تاریخ بدرستی به February 29 (29 فوریه) منتقل شده، افزایش داده است. سپس خط 22 یک شی **Date** بنام **d4** ایجاد کرده است که با تاریخ July 13,2002 مقداردهی اولیه شده است. سپس خط 26 مبادرت به افزایش **d4** به میزان 1 توسط عملگر پیش‌افزایشی سربارگذاری شده کرده است. خطوط 27-24 مبادرت به چاپ **d4** قبل و بعد از انجام پیش‌افزایش می‌کنند تا نشاندهند که کار بدرستی انجام گرفته است. سرانجام خط 31 مقدار **d4** را توسط عملگر پس‌افزایشی سربارگذاری شده افزایش داده است. خطوط 32-29 مبادرت به چاپ **d4** قبل و بعد از عملیات پس‌افزایشی کرده‌اند.

سربارگذاری عملگر افزایشی پیشوندی کار سر راستی است. عملگر پیش‌افزایشی (تعریف شده در شکل ۱۳-۱۱، خطوط 34-30) مبادرت به فراخوانی تابع کمکی یا یوتیلیتی **helpIncrement** برای افزایش تاریخ کرده است (تعریف شده در شکل ۱۳-۱۱، خطوط 93-76). عملگرد این تابع همانند یک مراقب یا حامل است و زمانیکه مبادرت به افزودن آخرین روز از ماه می‌کنیم، وارد صحنه می‌شود. اگر در ماه 12 قرار داشته باشیم، پس بایستی سال نیز افزایش داده شود و ماه با 1 تنظیم گردد. تابع **helpIncrement** از تابع **endOfMonth** برای افزایش صحیح روز استفاده می‌کند.

۱۳-۱۱ کلاس string از کتابخانه استاندارد

در این فصل، آموختید که می‌توانید یک کلاس **String** ایجاد کنید (شکل‌های ۹-۱۱ الی ۱۱-۱۱) که بسیار بهتر از رشته‌های * **char** به سبک C بوده و توسط C++ بکار گرفته شده‌اند. همچنین آموختید که

می‌توانید یک کلاس **Array** ایجاد کنید (شکل‌های ۶-۱۱ الی ۸-۱۱) که عملکردی بهتر از آرایه مبتنی بر اشاره‌گر به سبک C دارد و توسط C++ بکار گرفته شده‌اند.

ایجاد کلاس‌های سودمند با قابلیت استفاده مجدد همانند **String** و **Array** کاری زمان‌بر است. بطوریکه چنین کلاس‌های برای اینکه بتوانند توسط شما، دانشگاه، شرکت خودتان، سایر شرکت‌ها یا یک مجموعه علمی و صنعتی بکار گرفته شوند بایستی تست و خطایابی دقیق شده باشند. طراحان C++ دقیقاً اینکار را انجام داده‌اند و کلاس‌های **String** و **Vector** را همراه C++ استاندارد کرده‌اند. این کلاس‌ها در دسترس هر کسی که مشغول ایجاد برنامه‌های کاربردی با C++ است، قرار دارند.

برای به پایان بردن این فصل، دوباره به سراغ مثال **String** خود می‌رویم (شکل‌های ۹-۱۱ الی ۱۱-۱۱) و این بار آنرا با کلاس استاندارد **string** پیاده‌سازی می‌کنیم. همان وظایف مثال قبلی را نیز در این مثال اعمال می‌کنیم (البته با استفاده از قابلیت‌های کلاس استاندارد **string**). همچنین به بررسی تابع عضو از کلاس استاندارد **string** بنام‌های **empty** و **at** خواهیم پرداخت که بخشی از مثال **String** ما نبودند. تابع **empty** تعیین می‌کند که آیا رشته تهی است یا خیر. تابع **substr** رشته‌ای برگشت می‌دهد که بخشی از رشته موجود بوده و تابع **at** کاراکتر قرار گرفته در شاخص تعیین شده در رشته را برگشت می‌دهد (البته پس از بررسی قرار داشتن شاخص در محدودهٔ). در فصل هیجدهم به تفصیل به بررسی کلاس **string** خواهیم پرداخت.

کلاس string از کتابخانه استاندارد

برنامه شکل ۱۵-۱۱ پیاده‌سازی مجددی از برنامه ۱۱-۱۱ با استفاده از کلاس استاندارد **string** است. همانطوری که در این مثال خواهید دید، کلاس **string** تمام قابلیت‌های کلاس **String** ما را که در برنامه‌های ۹-۱۱ و ۱۰-۱۱ عرضه شده بودند، دارا است. کلاس **string** در سرآیند <**string**> تعریف شده (خط 7) و متعلق به فضای نامی **std** است (خط 8).

```
1  // Fig. 11.15: fig11_15.cpp
2  // Standard Library string class test program.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <string>
8  using std::string;
9
10 int main()
11 {
12    string s1( "happy" );
13    string s2( " birthday" );
14    string s3;
15
16    // test overloaded equality and relational operators
17    cout << "s1 is \"" << s1 << "\"; s2 is \"" << s2
18      << "\"; s3 is \"" << s3 << '\"'
19       << "\n\nThe results of comparing s2 and s1:"
20       << "\ns2 == s1 yields " << ( s2 == s1 ? "true" : "false" )
21       << "\ns2 != s1 yields " << ( s2 != s1 ? "true" : "false" )
```

```
22       << "\ns2 >  s1 yields " << ( s2 > s1 ? "true" : "false" )
23       << "\ns2 <  s1 yields " << ( s2 < s1 ? "true" : "false" )
24       << "\ns2 >= s1 yields " << ( s2 >= s1 ? "true" : "false" )
25       << "\ns2 <= s1 yields " << ( s2 <= s1 ? "true" : "false" );
26
27    // test string member function empty
28    cout << "\n\nTesting s3.empty():" << endl;
29
30    if ( s3.empty() )
31    {
32       cout << "s3 is empty; assigning s1 to s3;" << endl;
33       s3 = s1; // assign s1 to s3
34       cout << "s3 is \"" << s3 << "\"";
35    } // end if
36
37    // test overloaded string concatenation operator
38    cout << "\n\ns1 += s2 yields s1 = ";
39    s1 += s2; // test overloaded concatenation
40    cout << s1;
41
42    // test overloaded string concatenation operator with C-style string
43    cout << "\n\ns1 += \" to you\" yields" << endl;
44    s1 += " to you";
45    cout << "s1 = " << s1 << "\n\n";
46
47    // test string member function substr
48    cout << "The substring of s1 starting at location 0 for\n"
49       << "14 characters, s1.substr(0, 14), is:\n"
50       << s1.substr( 0, 14 ) << "\n\n";
51
52    // test substr "to-end-of-string" option
53    cout << "The substring of s1 starting at\n"
54       << "location 15, s1.substr(15), is:\n"
55       << s1.substr( 15 ) << endl;
56
57    // test copy constructor
58    string *s4Ptr = new string( s1 );
59    cout << "\n*s4Ptr = " << *s4Ptr << "\n\n";
60
61    // test assignment (=) operator with self-assignment
62    cout << "assigning *s4Ptr to *s4Ptr" << endl;
63    *s4Ptr = *s4Ptr;
64    cout << "*s4Ptr = " << *s4Ptr << endl;
65
66    // test destructor
67    delete s4Ptr;
68
69    // test using subscript operator to create lvalue
70    s1[ 0 ] = 'H';
71    s1[ 6 ] = 'B';
72    cout << "\ns1 after s1[0] = 'H' and s1[6] = 'B' is: "
73       << s1 << "\n\n";
74
75    // test subscript out of range with string member function "at"
76    cout << "Attempt to assign 'd' to s1.at( 30 ) yields:" << endl;
77    s1.at( 30 ) = 'd'; // ERROR: subscript out of range
78    return 0;
79 } // end main
```

```
s1 is "happy"; s2 is " birthday"; s3 is ""

The results of comparing s2 and s1:
s2 == s1 yields false
s2 != s1 yields true
s2 >  s1 yields false
s2 <  s1 yields true
s2 >= s1 yields false
s2 <= s1 yields true

Testing s3.empty:
```

```
s3 is empty; assiging s1 to s3;
s3 is "happy"

s1 += s2 yields s1 = happy birthday

s1 += " to you" yields
s1 = happy birthday to you

The substring of s1 starting at location 0 for
14 characters, s1.substr(0, 14), is:
happy birthday

The substring of s1 starting at
location 15, s1.substr(15), is:
to you

*s4Ptr = happy birthday to you

assigning *s4Ptr to *s4Ptr
*s4Ptr = happy birthday to you
Destructor: happy birthday to you

s1 after s1[0] = 'H'and s1[6] = 'B'is: Happy Birthday to you

Attempt to assign 'd' to s1[30] yields:
abnormal program termination
```

شکل ۱۵-۱۱ کلاس string از کتابخانه استاندارد.

خطوط 14-12 سه شی **string** ایجاد می‌کنند. **S1** با کلمه "happy"، **s2** با کلمه "birthday" و **s3** با استفاده از سازنده پیش‌فرض رشته به منظور ایجاد یک رشته تهی. مقداردهی اولیه می‌شوند. خطوط 18-17 این سه شی را با استفاده از **cout** و عملگر >>، که طراحان کلاس **string** آنرا برای کار با شی‌های رشته سربارگذاری کرده‌اند، چاپ می‌کنند. سپس خطوط 25-19 نتایج حاصله از مقایسه **s2** با **s1** را با استفاده از عملگرهای برابری و رابطه‌ای سربارگذاری شده کلاس **string**، بنمایش در می‌آورند.

کلاس **String** ما (شکل‌های ۹-۱۱ الی ۱۰-۱۱) یک عملگر **operator!** سربارگذاری شده تدارک دیده بود که مبادرت به تست یک رشته می‌کرد که آیا تهی است یا خیر. کلاس استاندارد **string** فاقد این قابلیت بعنوان یک عملگر سربارگذاری شده است و بجای آن دارای تابع عضو **empty** می‌باشد که در خط 30 از آن استفاده کرده‌ایم. اگر رشته تهی باشد، تابع **empty** مقدار **true** و در غیر اینصورت **false** برگشت خواهد داد.

خط 33 به توضیح عملگر تخصیص سربارگذاری شده کلاس **string** پرداخته که در آن **s1** به **s3** تخصیص می‌یابد. خط 34 برای نشان دادن اینکه عملیات تخصیص بدرستی صورت گرفته، **s3** را چاپ می‌کند.

خط 39 به توضیح عملگر =+ سربارگذاری شده کلاس **string** به منظور اتصال رشته پرداخته است. در این مورد، محتویات **s2** به **s1** مرتبط می‌شوند. سپس خط 40 نتیجه این رشته را که در **s1** ذخیره شده است، چاپ می‌کند.

کلاس **String** ما (شکل‌های ۹-۱۱ و ۱۰-۱۱) حاوی عملگر سربارگذاری شده **()operator** به منظور تهیه زیر رشته بود. کلاس استاندارد **string** فاقد چنین قابلیتی با عملگر سربارگذاری شده است و بجای آن دارای تابع عضو **substr** می‌باشد (خطوط 50 و 55). با فراخوانی **substr** در خط 50 یک زیر رشته 14 کاراکتری (آرگومان اول) از **s1** با موقعیت آغازین صفر (آرگومان دوم) بدست می‌آید. با فراخوانی **substr** در خط 55 یک زیر رشته از موقعیت آغازین 15 رشته **s1** تهیه می‌شود. زمانیکه آرگومان دوم مشخص نشود، **substr** مبادرت به برگشت دادن مابقی رشته از آنجایی خواهد کرد که فراخوانی گردیده است.

خط 58 بصورت دینامیکی یک شی **string** اخذ کرده و آنرا با کپی از **s1** مقداردهی اولیه می‌کند. نتیجه اینکار فراخوانی سازنده کپی کننده کلاس **string** است. خط 63 از عملگر سربارگذاری شده = کلاس **string** برای تست وضعیت خود تخصیصی استفاده کرده است.

خطوط 71-70 از عملگر سربارگذاری شده **[]** کلاس **string** به منظور ایجاد *lهاlvalue* استفاده کرده‌اند تا بتوان کاراکترهای جدید را با کاراکترهای موجود در **s1** جایگزین کرد. خط 73 مقدار جدید **s1** را چاپ می‌کند. در کلاس **String** ما، عملگر سربارگذاری شده **[]** وظیفه بررسی مرزها را انجام می‌داد تا مشخص شود که آیا شاخص دریافتی به عنوان یک آرگومان، یک شاخص معتبر است یا خیر. اگر شاخص معتبر نبود، عملگر یک پیغام خطا چاپ می‌کرد و برنامه خاتمه می‌پذیرفت. عملگر سربارگذاری شده **[]** کلاس استاندارد **string** عملیات بررسی مرزها را انجام می‌دهد. از اینرو بایستی برنامه‌نویس مطمئن گردد که عملیات استفاده کننده از عملگر سربارگذاری شده **[]** کلاس استاندارد **string** ناخواسته در عناصر خارج از مرزهای رشته، تغییری را حادث نسازد. کلاس استاندارد **string** با استفاده از تابع عضو خود بنام **at** بررسی مرزها را انجام می‌دهد و در صورتیکه شاخص معتبر نباشد، یک «استثنا به راه» می‌اندازد. در چنین وضعیتی، در حالت پیش‌فرض برنامه C++ خاتمه می‌پذیرد. اگر شاخص معتبر باشد، تابع **at** کاراکتر را از مکان تعیین شده بعنوان یک *lvalue* تغییرپذیر یا یک *lvalue* ثابت و با توجه به محتویات فراخوانی برگشت می‌دهد. خط 77 نشان می‌دهد که فراخوانی تابع **at** بر روی یک شاخص نامعتبر صورت گرفته است.

۱۱-۱٤ سازنده‌های صریح

در بخش‌های ۸-۱۱ و ۹-۱۱ در ارتباط با سازنده تک آرگومانی صحبت کردیم که می‌توانست توسط کامپایلر به منظور انجام یک تبدیل ضمنی بکار گرفته شود. عملیات بصورت اتوماتیک صورت می‌گیرد و برنامه‌نویس نیازی به استفاده از یک عملگر تبدیل کننده ندارد. در برخی از شرایط تبدیلات ضمنی مناسب

نبوده یا زمینه‌ساز خطا می‌شوند. برای مثال کلاس **Array** ما در شکل ۶-۱۱ سازنده‌ای تعریف کرده است که یک آرگومان از نوع **int** دیافت می‌کند. هدف از این سازنده ایجاد یک شی **Array** حاوی تعدادی عنصر تعیین شده توسط آرگومان **int** است. با این وجود، این سازنده می‌تواند توسط کامپایلر در انجام یک تبدیل ضمنی، درست بکار گرفته نشود.

استفاده تصادفی از یک سازنده تک آرگومانی بعنوان یک سازنده تبدیل کننده

برنامه شکل ۱۶-۱۱ از کلاس **Array** شکل‌های ۶-۱۱ و ۷-۱۱ به منظور نشان دادن یک تبدیل ضمنی غلط استفاده کرده است. خط 13 در **main** مبادرت به نمونه‌سازی شی **integers1** و فراخوانی سازنده تک آرگومانی به مقدار صحیح 7 که تعداد عناصر در آرایه را تعیین می‌کند، کرده است. از شکل ۷-۱۱ بخاطر دارید که سازنده **Array** یک آرگومان **int** برای مقداردهی اولیه تمام عناصر آرایه با صفر دریافت می‌کرد. خط 14 تابع **outputArray** (تعریف شده در خطوط 24-20) را که بعنوان آرگومان یک **const Array &** به یک **Array** دریافت می‌کند، فراخوانی می‌نماید. خروجی تابع، تعداد عناصر در آرگومان **Array** و محتویات آن است. در این مورد، سایز آرایه 7 است، و از اینرو هفت، صفر در خروجی قرار دارد.

خط 15 تابع **outputArray** را با مقدار 3 بعنوان آرگومان فراخوانی کرده است. با این وجود، این برنامه فاقد یک تابع بنام **outputArray** است که یک آرگومان صحیح (**int**) دیافت می‌کند. از اینرو کامپایلر تعیین می‌کند که آیا کلاس **Array** یک سازنده تبدیل کننده تدارک دیده است که بتواند یک **int** را به یک **Array** تبدیل کند یا خیر. هر سازنده‌ای که یک آرگومان منفرد دریافت کند بعنوان یک سازنده تبدیل کننده در نظر گرفته می‌شود و کامپایلر فرض می‌کند سازنده **Array** که یک **int** دریافت می‌نماید یک سازنده تبدیل کننده است و از آن برای تبدیل آرگومان 3 به یک شی **Array** موقت که حاوی سه عنصر است، استفاده می‌کند. سپس کامپایلر مبادرت به ارسال شی **Array** موقت به تابع **outputArray** می‌کند تا محتویات آنرا چاپ کند. بنابر این، حتی در زمانیکه بصورت صریح یک تابع **outputArray** تدارک ندیده باشیم که یک آرگومان **int** دریافت می‌کند، کامپایلر قادر به کامپایل خط 15 است. خروجی برنامه نمایشی از محتویات آرایه سه عنصری حاوی صفرها است.

```
1  // Fig. 11.16: Fig11_16.cpp
2  // Driver for simple class Array.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Array.h"
8
9  void outputArray( const Array & ); // prototype
10
11 int main()
12 {
```

```
13    Array integers1( 7 ); // 7-element array
14    outputArray( integers1 ); // output Array integers1
15    outputArray( 3 ); // convert 3 to an Array and output Array's contents
16    return 0;
17 }  // end main
18
19 // print Array contents
20 void outputArray( const Array &arrayToOutput )
21 {
22    cout << "The Array received has " << arrayToOutput.getSize()
23       << " elements. The contents are:\n" << arrayToOutput << endl;
24 } // end outputArray
```

```
The Array received has 7 elements. The contents are:
           0           0           0           0
           0           0           0

The Array received has 3 elements. The contents are:
           0           0           0
```

شکل ۱۶-۱۱ | سازنده‌های تک آرگومانی و تبدیلات ضمنی.

اجتناب از استفاده تصادفی از یک سازنده تک آرگومانی بعنوان یک سازنده تبدیل کننده

زبان C++ دارای کلمه کلیدی **explicit** به منظور متوقف کردن تبدیلات ضمنی از طریق سازنده‌های تبدیل کننده در مواردی است که اجازه انجام چنین تبدیلاتی وجود ندارد. سازنده‌ای که بصورت **explicit** اعلان شده است نمی‌تواند در یک تبدیل ضمنی بکار گرفته شود. در شکل ۱۷-۱۱ یک سازنده **explicit** در کلاس **Array** اعلان شده است. تنها تغییر اعمال شده در **Array.h** افزودن کلمه کلیدی **explicit** به اعلان سازنده تک آرگومانی در خط 15 است.

```
1  // Fig. 11.17: Array.h
2  // Array class for storing arrays of integers.
3  #ifndef ARRAY_H
4  #define ARRAY_H
5
6  #include <iostream>
7  using std::ostream;
8  using std::istream;
9
10 class Array
11 {
12    friend ostream &operator<<( ostream &, const Array & );
13    friend istream &operator>>( istream &, Array & );
14 public:
15    explicit Array( int = 10 ); // default constructor
16    Array( const Array & ); // copy constructor
17    ~Array(); // destructor
18    int getSize() const; // return size
19
20    const Array &operator=( const Array & ); // assignment operator
21    bool operator==( const Array & ) const; // equality operator
22
23    // inequality operator; returns opposite of == operator
24    bool operator!=( const Array &right ) const
25    {
26       return ! ( *this == right ); // invokes Array::operator==
27    } // end function operator!=
28
29    // subscript operator for non-const objects returns lvalue
30    int &operator[]( int );
31
32    // subscript operator for const objects returns rvalue
33    const int &operator[]( int ) const;
34 private:
```

```
35    int size; // pointer-based array size
36    int *ptr; // pointer to first element of pointer-based array
37 }; // end class Array
38
39 #endif
```

شکل ۱۷-۱۱ | تعریف کلاس Array با سازنده explicit.

نیازی به تغییر در فایل کد منبع حاوی تعریف تابع عضو کلاس **Array** نیست. برنامه شکل ۱۸-۱۱ نمایشی از نسخه اصلاح شده برنامه ۱۶-۱۱ است.

زمانیکه این برنامه کامپایل شود، کامپایلر یک پیغام خطا مبنی بر اینکه مقدار صحیحی به **outputArray** در خط 15 ارسال شده و نمی‌تواند به یک **const Array &** تبدیل شود، صادر می‌کند. پیغام خطای کامپایلر در پنجره خروجی نشان داده شده است. خط 61 نشان می‌دهد که چگونه می‌توان از سازنده **explicit** در ایجاد یک آرایه موقت از 3 عنصر و ارسال آن به تابع **outputArray** استفاده کرد.

خطای برنامه‌نویسی

*اقدام به احضار سازنده **explicit** برای یک تبدیل ضمنی، خطای کامپایل بدنبال خواهد داشت.*

خطای برنامه‌نویسی

*استفاده از **explicit** در کنار اعضای داده یا توابع عضو بجز سازنده تک آرگومانی خطای کامپایل است.*

```
1  // Fig. 11.18: Fig11_18.cpp
2  // Driver for simple class Array.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Array.h"
8
9  void outputArray( const Array & ); // prototype
10
11 int main()
12 {
13    Array integers1( 7 ); // 7-element array
14    outputArray( integers1 ); // output Array integers1
15    outputArray( 3 );// convert 3 to an Array and output Array's contents
16    outputArray( Array( 3 ) );//explicit single-argument constructor call
17    return 0;
18 }  // end main
19
20 // print array contents
21 void outputArray( const Array &arrayToOutput )
22 {
23    cout << "The Array received has " << arrayToOutput.getSize()
24       << " elements. The contents are:\n" << arrayToOutput << endl;
25 } // end outputArray
```

```
C:\cpphtp5_examples\ch11\Fig11_17_18\Fig11_18.cpp(15) : error C2664:
   'outputArray': cannot convert parameter 1 from 'int'to 'const Array &'
       Reason: cannot convert from 'int'to 'const Array'
       Constructor for class 'Array' is declared 'explicit'
```

شکل ۱۸-۱۱ | توصیف عملکرد سازنده explicit.

فصل
دوازدهم

برنامه‌نویسی شی‌گرا: توارث

اهداف

- ایجاد کلاس‌ها با ارث‌بری از کلاس‌های جدید.
- نحوه استفاده مجدد از نرم‌افزار توسط توارث.
- مفهوم کلاس‌های مبنا و کلاس‌های مشتق شده و رابطه مابین آنها.
- عضو تصریح کننده دسترسی protected.
- کاربرد سازنده‌ها و نابود کننده‌ها در سلسله مراتب توارث.
- تفاوت مابین توارث public، protected و private.
- کاربرد توارث در بهینه‌سازی نرم‌افزارهای موجود.

رئوس مطالب

۱-۱۲ مقدمه

در این فصل، بحث خود را با معرفی یکی از ویژگی‌های مهم برنامه‌نویسی شی‌گرا (OOP) یعنی توارث یا ارث‌بری آغاز می‌کنیم. توارث فرمی از بکارگیری مجدد نرم‌افزار است که در آن کلاس‌های ایجاد شده موجودیت و رفتار خود را براساس اطلاعات یک کلاس موجود بدست آورده و در صورت نیاز حاوی قابلیت‌های جدید هستند. بکارگیری مجدد نرم‌افزار سبب کاهش مدت زمان توسعه نرم‌افزار شده و کیفیت آنرا بطور موثری افزایش می‌دهد.

به هنگام ایجاد یک کلاس، بجای نوشتن کامل متغیرهای نمونه و متدها، برنامه‌نویس می‌تواند تعیین کند که کلاس جدید بایستی متغیرها، خصوصیات و متدهای کلاس را از یک کلاس دیگر به ارث ببرد. کلاسی که قبلاً تعریف شدهٔ، کلاس *مبنا* نامیده می‌شود و کلاس جدید بعنوان یک کلاس *مشتق شده* شناخته می‌شود. (در زبان‌های برنامه‌نویسی دیگری همانند جاوا، به کلاس مبنا، *سوپرکلاس* و کلاس مشتق شدهٔ، *زیرکلاس* گفته می‌شود.) پس از ایجاد کلاس، هر کلاس مشتق شده می‌تواند تبدیل به یک کلاس مبنا برای کلاس‌هایی شود که بعدها از آن مشتق خواهند شد. یک کلاس مشتق شده که دارای متغیرها، خصوصیات و متدهای منحصر بفرد است، معمولاً بزرگتر از کلاس مبنای خود می‌باشد. از اینرو، یک کلاس مشتق شده به نسبت کلاس مبنای خود تخصصی‌تر است و نشاندهنده گروهی خاص و مرتبط از

شی‌ها است. عموماً یک کلاس مشتق شدهٔ حاوی رفتار کلاس مبنای خود به همراه قابلیت‌ها و رفتارهای دیگر است. یک کلاس مبنای *مستقیم*، کلاس مبنائی است که کلاس‌های مشتق شده بصورت صریح از آن ارث بری دارند. یک کلاس مبنای *غیرمستقیم*، از دو یا بیش از چند سطح سلسله مراتب کلاس ارث بری دارد. در توارث یگانه، یک کلاس از یک کلاس مبنا مشتق می‌شود. C++ از توارث چندگانه پشتیبانی می‌کند. توارث چندگانه زمانی اتفاق می‌افتد که یک کلاس از بیش از یک کلاس مبنا مشتق شود.

زبان **C++** سه نوع توارث یا ارث‌بری را پیشنهاد می‌کند: **public**، **protected** و **private** در این فصل بحث ما متمرکز بر روی ارث‌بری **public** بوده و تا حدودی به توضیح دو نوع دیگر هم خواهیم پرداخت. در فصل ۲۱، ساختمان‌های داده، نشان خواهیم داد که چگونه ارث‌بری **private** می‌تواند بعنوان جانشینی برای ترکیب بکار گرفته شود. از فرم سوم، یعنی ارث‌بری **protected** بندرت استفاده می‌شود. در ارث‌بری **public**، هر شی از یک کلاس مشتق شده یک شی از کلاس مبنای مشتق شده نیز محسوب می‌شود. با این وجود، شی‌های کلاس مبنا، شی‌های از کلاس‌ها مشتق شده از خودشان نیستند. برای مثال اگر وسیله نقلیه یک کلاس مبنا باشد و اتومبیل یک کلاس مشتق شده، پس تمام اتومبیل‌ها، وسیله نقلیه محسوب می‌شوند، اما تمام وسایل نقلیه اتومبیل نیستند. همانطوری که به آموزش برنامه‌نویسی شی‌گرا در فصل ۱۲ و ۱۳ ادامه می‌دهیم از مزایای موجود در این روابط در انجام کارهای جالب توجه استفاده خواهیم کرد.

تجربه ایجاد سیستم‌های نرم‌افزاری نشان داده است که بخش قابل توجهی از کد در ارتباط با حل موارد خاص هستند. زمانیکه برنامه‌نویسان گرفتار موارد خاص می‌شوند، جزئیات کار می‌تواند کل موضوع را مبهم سازد. با برنامه‌نویسی شی‌گرا، تمرکز برنامه‌نویسان بر روی نقاط مشترک شی‌ها در سیستم است بجای اینکه به موارد خاص متکی باشند.

مابین رابطه *است–یک* (*is-a*) و رابطه *دارد–یک* (*has-a*) تفاوت قائل هستیم. رابطه is-a نشاندهنده توارث است. در این رابطه با یک شی از یک کلاس مشتق شده می‌توان بعنوان یک شی از کلاس مبنا خود رفتار کرد، برای مثال اتومبیل یک وسیله است (رابطه است–یک)، از اینرو خصوصیات و رفتار یک وسیله نقلیه، خصوصیات اتومبیل هم محسوب می‌شوند. در مقابل رابطه has-a قرار دارد که نشاندهنده ترکیب می‌باشد. (ترکیب در فصل ۱۰ توضیح داده شده است). در رابطه has-a، یک شی حاوی یک یا چندین شی از کلاس‌های دیگر بعنوان عضو است. برای مثال، یک اتومبیل دارای کامپونت‌های متعددی است، دارای چرخ، پدال گاز، موتور و اجزای دیگر است.

ممکن است توابع عضو کلاس مشتق شده نیازمند دسترسی به اعضای داده و توابع عضو کلاس مبنای خود داشته باشند. یک کلاس مشتق شده می‌تواند به اعضای غیر ***private***، کلاس مبنای خود دسترسی

داشته باشد. اعضای کلاس مبنا که قادر به دستیابی به توابع عضو یک کلاس مشتق شده از کلاس مبنا به طریق توارث نیستند، باید بصورت ***private*** در کلاس مبنا اعلان شوند.

مهندسی نرم‌افزار

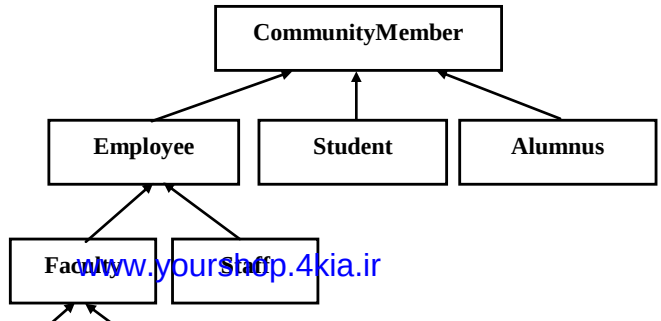
توابع عضو یک کلاس مشتق شده نمی‌توانند بصورت مستقیم به اعضای ***private*** کلاس مبنا دسترسی یابند.

یکی از مشکلاتی که در توارث وجود دارد این است که کلاس مشتق شده داده‌های عضو و توابع عضوی را که به آنها نیاز ندارد به ارث می‌برد. این وظیفه طراح کلاس است تا مطمئن شود قابلیت‌های تدارک دیده شده توسط کلاس، مناسب، کلاس‌های هستند که بعدها از آن مشتق خواهند شد. حتی در زمانیکه خصیصه یا متد کلاس مبنا برای کلاس‌های مشتق شده مناسب طراحی شده باشند، گاهاً کلاس‌های مشتق شده نیاز به توابع یا خصوصیات خاص خود دارند تا وظیفه خود را به انجام برسانند. در چنین مواردی، تابع عضو کلاس مبنا می‌تواند در کلاس مشتق شده بازنویسی (تعریف مجدد) شود.

۱۲-۲ کلاس‌های مبنا و کلاس‌های مشتق شده

غالباً یک شی از یک کلاس، به همان اندازه شیی از یک کلاس دیگر است. برای مثال در علم هندسه، یک مستطیل یک چهار ضلعی است. از اینرو می‌توان گفت که کلاس **Rectangle** از کلاس **Quadrilateral** ارث‌بری داشته است. در اینحالت کلاس **Quardrilateral** کلاس مبنا است و کلاس **Rectangle** کلاس مشتق شده از آن می‌باشد. مستطیل نوع خاصی از چهار ضلعی است، اما تصور اشتباهی است که بگوییم که یک چهارضلعی یک مستطیل است، چرا که چهارضلعی می‌تواند یک متوازی‌الاضلاع یا نوع دیگری از **Quardrilaterd** باشد. در جدول شکل ۱-۱۲ لیستی از چند مثال ساده در ارتباط با کلاس‌های مبنا و کلاس‌های مشتق شده، به نمایش درآمده است.

کلاس مبنا	کلاس‌های مشتق شده
Student	GranduateStudent
	UndergraduateStudent
Shape	Circle
	Triangle
	Rectangle
	Sphere
	Cube
Loan	CarLoan
	HomeImprovementLoan
	MortgageLoan



Employee	Faculty Staff
Account	CheckingAccount SavingsAccount

شکل ۱-۱۲ مثال‌های از توارث.

بدلیل اینکه هر شی از کلاس مشتق شده، شی از کلاس مبنای خود است و یک کلاس مبنا می‌تواند تعداد زیادی کلاس مشتق شده داشته باشد، از اینرو، مجموعه شی‌های به نمایش در آمده توسط کلاس مبنا بیشتر از مجموعه شی‌های عرضه شده توسط هر کلاس مشتق شده از خود کلاس مبنا است. برای مثال، کلاس مبنای **Vehicle** نشاندهنده تمام وسایل نقلیه، شامل اتومبیل‌ها، کامیون‌ها، دوچرخه‌ها و غیره است. در مقابل، کلاس مشتق شده **Car** فقط نشاندهنده زیر مجموعه کوچکی از تمام **Vehicle** (وسایل نقلیه) است.

رابطه توارث را می‌توان به فرم یک سلسله مراتب درختی به نمایش در آورد. موجودیت یک کلاس در رابطه توارث با کلاس‌های مشتق شده آن مشخص می‌گردد. اگر چه کلاس‌ها می‌توانند موجودیت‌های مستقلی داشته باشند، اما زمانیکه در ترتیبات توارثی بکار گرفته می‌شوند، با کلاس‌های دیگر مرتبط می‌گردند.

اجازه دهید تا به بررسی و ایجاد یک سلسله مراتب توارث ساده در پنج سطح بپردازیم (عرضه شده با دیاگرام کلاس UML در شکل ۲-۱۲). یک جامعه دانشگاهی را با صدها عضوی که دارد در نظر بگیرید. این اعضا متشکل از کارمندان، فارغ‌التحصیلان و دانشجویان هستند. کارمندان می‌توانند اعضای هیت علمی باشند یا کارمند ساده. اعضای هیت علمی می‌توانند، مدیر یا استاد باشند. با این وجود، برخی از مدیران می‌توانند در کلاس‌ها تدریس کنند. دقت کنید که از توارث مضاعف به فرم **AdministratorTeacher** استفاده کرده‌ایم. این ساختار سازماندهی، نمایانگر یا سلسله مراتب توارث است و در شکل ۲-۱۲ دیده می‌شود. دقت کنید که سلسله مراتب توارث می‌تواند حاوی کلاس‌های دیگری نیز باشد. برای مثال، دانشجویان می‌توانند، در زمرهٔ دانشجویان فارغ‌التحصیل یا دانشجویان فارغ‌التحصیل نشده قرار گیرند.

شکل ۲-۱۲ | سلسله مراتب توارث برای کلاس CommunityMembers.

در هر فلش این سلسله مراتب، رابطه وجود داشتن برقرار است. برای مثال، اگر فلش‌ها را دنبال کنیم، متوجه می‌شویم که **Employee** یک **CommunityMember** است یا **Teacher** یک عضو **Faculty** است. در واقع **CommunityMember**، کلاس مبنا مستقیم برای **Employee**، **Student** و **Alumnus** است. علاوه بر این، **CommunityMember** یک کلاس مبنای غیرمستقیم برای تمام دیگر کلاس‌ها در دیاگرام سلسله مراتب است.

اگر از پایین دیاگرام حرکت کنیم و جهت فلش‌ها را دنبال نمائیم به کلاس مبنا در بالاترین سطح می‌رسیم. برای مثال، یک **AdministratorTeacher** یک **Administrator** بوده، عضو **Faculty** و **Employee** و **CommunityMembers** است

حال به سلسله مراتب توارث **Shape** در شکل ۳-۱۲ توجه کنید. این سلسله مراتب با کلاس مبنای **Shape** آغاز می‌شود. کلاس‌های **TwoDimensionalShape** (شکل‌های دوبعدی) و **ThreeDimensionalShape** (شکل‌های سه‌بعدی) از کلاس مبنای **Shape** مشتق شده‌اند. شکل‌ها یا دوبعدی یا سه‌بعدی هستند. سطح سوم این سلسله مراتب حاوی برخی از انواع مشخص از اشکال دوبعدی و سه‌بعدی است. همانطوری که در شکل ۲-۱۲ می‌توانستیم فلش‌ها را از پایین دیاگرام دنبال کرده و به کلاس مبنا در بالاترین سطح برسیم، در این سلسله مراتب کلاس، چندین رابطه **is-a** وجود دارد. برای نمونه، یک **Triangle** (مثلث) یک شکل دوبعدی و یک شکل است **(shape)**، در حالیکه یک **Sphere** (کره) یک شکل سه‌بعدی و یک شکل است. توجه کنید که این سلسله مراتب می‌توانست حاوی کلاس‌های دیگری همانند مستطیل‌ها، بیضی‌ها و ذوزنقه‌ها باشد که همگی شکل‌های دوبعدی هستند.

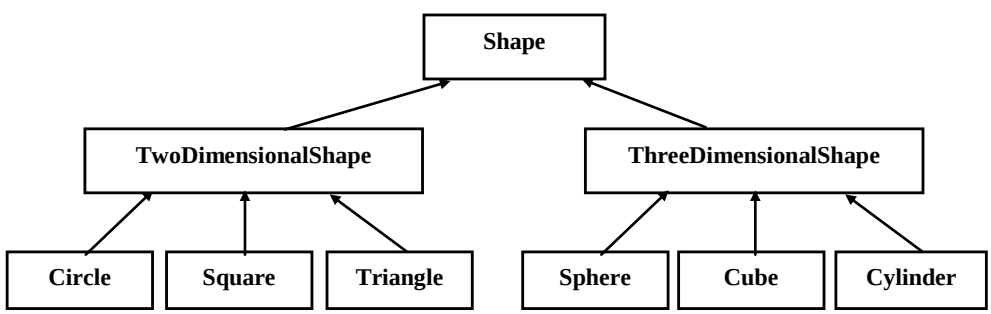
برای تصریح اینکه کلاس **TwoDimensionalShpae** از کلاس **Shape** مشتق شده (یا از آن ارث بری دارد)، بایستی کلاس **TwoDimensionalShape** در C++ بصورت زیر تعریف شود:

```
class TwoDimensionalShape :public Shape
```

عبارت فوق مثالی از توارث سراسری یا **public** است، که در اکثر مواقع بکار گرفته می‌شود. در توارث، اعضای **private** از یک کلاس مبنا بصورت مستقیم از طریق کلاس‌های مشتق شده در دسترس

نمی‌باشند، اما هنوز هم این اعضای **private** از کلاس مبنا به ارث برده می‌شوند. تمام دیگر اعضای کلاس مبنا، عضو دسترسی اصلی خود را به هنگام تبدیل شدن به اعضای کلاس مشتق شده، حفظ و نگهداری می‌کنند (برای مثال، اعضای **public** در کلاس مبنا، تبدیل به اعضای **public** در کلاس مشتق شده می‌شوند و همانطوری که بزودی خواهید دید، اعضای **protected** در کلاس مبنا، تبدیل به اعضای **protected** در کلاس مشتق شده خواهند شد). در میان این اعضای کلاس مبنای به ارث برده شده، کلاس مشتق شده می‌تواند در اعضای **private** کلاس مبنا دستکاری نماید (اگر این اعضای به ارث برده شده چنین قابلیتی در کلاس مبنا داشته باشند). امکان تلقی کردن شی‌های کلاس مبنا و شی‌های کلاس مشتق شده بطریقه مشابه وجود دارد.

در فصل دهم بطور اجمال در مورد رابطه وجود داشتن بحث کردیم که در آن کلاس‌ها اعضای داشتند کهشی‌های از کلاس‌های دیگر بودند. چنین روابطی با بکارگیری ترکیب از کلاس‌های موجود، اقدام به ایجاد کلاس‌ها می‌کنند. برای مثال، گفتن اینکه کلاس **Employee** از کلاس **BirthDate** یا از **TelephoneNumber** است، کاملاً اشتباه می‌باشد. با این وجود، مناسب خواهد بود که بگویم **Employee** دارای یک **BirthDate** است و اینکه **Employee** دارای **TelephoneNumber** می‌باشد.



شکل ۳-۱۲ | بخشی از سلسله مراتب کلاس Shape.

۳-۱۲ اعضای protected

در فصل سوم به توضیح اصلاح کننده‌های دسترسی **public** و **private** پرداختیم. اعضای یک کلاس **public** از هر کجای برنامه که دارای مراجعه‌ای به شی از کلاس مبنا یا یکی از کلاس‌های مشتق شده از آن است در دسترس هستند. اعضای یک کلاس مبنای **private** فقط در درون بدنه کلاس مبنا و دوستان

(friends) آن در دسترس می‌باشند. در این بخش، به توضیح عضو اصلاح کننده دسترسی دیگری بنام **protected** می‌پردازیم.

دسترسی **protected** عرضه کننده یک سطح حفاظتی میانی مابین دسترسی‌های **public** و **private** است. اعضای یک کلاس مبنای **protected** می‌توانند فقط در کلاس مبنا یا در هر کلاس مشتق شده از کلاس یا دوستان آن کلاس در دسترس قرار گیرند.

معمولاً متدهای کلاس مشتق شده می‌توانند بسادگی به اعضای **public** و **protected** کلاس مبنا با استفاده از اسامی اعضا مراجعه داشته باشند. هنگامی که یک تابع عضو از کلاس مشتق شده می‌خواهد به یک عضو کلاس مبنا دسترسی یابد می‌تواند با قرار دادن نام عضو کلاس مبنا به همراه نام کلاس مبنا و عملگر باینری تفکیک قلمرو (::) اینکار را انجام دهد. در بخش ۴-۱۲ در ارتباط با دسترسی به اعضای مجدد تعریف شده از کلاس مبنا می‌پردازیم و در بخش ۴-۴-۱۲ از داده **protected** شده استفاده می‌کنیم.

٤-۱۲ ارتباط مابین کلاس‌های مبنا و کلاس‌های مشتق شده

در این بخش، از یک سلسله مراتب توارث که حاوی انواع کارمندان در برنامه پرداخت دستمزد یک شرکت است استفاده می‌کنیم تا به توضیح رابطه موجود مابین یک کلاس مبنا و یک کلاس مشتق شده بپردازیم. کارمندان کمیسیون (یا کارمندان حق‌العمل‌کار) که بعنوان شی‌های از کلاس مبنا عرضه خواهند شد، حقوق خود را بصورت درصدی از فروش دریافت می‌کنند، در حالیکه کارمندان کمیسیون مبتنی بر پایه حقوق (که بعنوان شی‌های از کلاس مشتق شده عرضه خواهند شد) یک حقوق پایه به همراه درصدی از فروش را دریافت می‌کنند. بحث خود را که در ارتباط با رابطه موجود مابین این دو نوع کارمند است به دقت و به کمک پنج مثال مطرح می‌کنیم:

۱- در اولین مثال، یک کلاس **CommissionEnployee** ایجاد می‌کنیم که حاوی اعضا داده **private** بعنوان نام، نام خانوادگی، شماره تامین اجتماعی، نرخ کمیسیون (درصد) و مبلغ ناخالص (یعنی مجموع) فروش است.

۲- در مثال دوم اقدام به تعریف کلاس **BasePlusCommissionEmployee** می‌کنیم که حاوی اعضای داده **private** بعنوان نام، نام خانوادگی، شماره تامین اجتماعی، نرخ کمیسیون، مبلغ ناخالص فروش و حقوق پایه است. این کلاس را با نوشتن خط به خط کدهای مورد نیاز کلاس ایجاد می‌کنیم. بزودی خواهید دید که ایجاد این کلاس به آسانی از طریق ارث‌بری از کلاس **CommissionEmployee** امکان‌پذیر است.

۳- در مثال سوم یک نسخه جدید از کلاس **BasePlusComissionEmployee** تعریف می‌کنیم که مستقیماً از کلاس **CommissionEmployee** ارث‌بری دارد و مبادرت به دسترسی به اعضای **private** این کلاس می‌کند، که نتیجه اینکار خطای کامپایل خواهد بود، چرا که کلاس مشتق شده نمی‌تواند به داده **private** (خصوصی) کلاس مبنا دسترسی پیدا کند.

۴- مثال چهارم نشان می‌دهد که اگر داده **CommissionEmployee** بصورت **protected** (حفاظت شده) اعلان شود، نسخه جدید کلاس **BasePlusCommissionEmployee** که از کلاس **CommissionEmpolyee** ارث‌بری دارد می‌تواند مستقیماً به داده آن دسترسی پیدا کند. به همین منظور، نسخه جدیدی از کلاس **CommssionEmployee** را با داده **protected** تعریف می‌کنیم. هر دو نسخه ارث‌بر و غیر ارث‌بر کلاس‌های **BasePlusCommissionEmployee** دارای قابلیت‌های یکسان هستند، اما نشان خواهیم داد که ایجاد و مدیریت نسخه ارث‌بر بسیار آسانتر است.

۵- پس از بحث در مورد قواعد استفاده از داده **protected**، مثال پنجمی ایجاد می‌کنیم که اقدام به تنظیم اعضای داده **CommissionEmployee** برای برگشت به حالت **private** می‌کند تا مهندسی نرم‌افزار مناسبی داشته باشیم. در این مثال نشان داده می‌شود که کلاس مشتق شده **BasePlusCommissionEmpolyee** می‌تواند توسط توابع **public** کلاس مبنا به منظور دستکاری کردن داده خصوصی **CommissionEmployee** بکار گرفته شود.

۱-٤-۱۲ ایجاد و استفاده از کلاس *ComissionEmployee*

اجازه دهید تا ابتدا به بررسی تعریف کلاس **ComissionEmployee** بپردازیم (شکل‌های ۴-۱۲ و ۵-۱۲) فایل سرآیند **CommissionEmolpyee** (شکل ۴-۱۲) مشخص کننده سرویس‌های سراسری کلاس **CommssionEmployee** است که شامل یک سازنده (خطوط 13-12) و توابع عضو **earnings** (خط 30) و **print** (خط 31) است.

در خطوط 28-15 توابع سراسری *get* و *set* برای کار با اعضای داده کلاس بنام **firstName**، **lastName**, **socialSecurityNumber** (شماره تامین اجتماعی)، **grossSales** (ناخالص فروش) و **commissonRate** (نرخ کمیسیون) است (اعلان شده در خطوط 77-33). فایل سرآیند **CommissionEmployee** مشخص می‌کند که هر یک از این اعضای داده حالت **private** (خصوصی) دارند، از اینرو شی‌های سایر کلاس‌ها نمی‌توانند مستقیماً به این داده دسترسی پیدا کنند. اعلان اعضای داده خصوصی و تدارک دیدن توابع *get* و *set* غیر خصوصی برای دستکاری کردن و اعتبارسنجی اعضا داده و به داشتن مهندسی نرم‌افزار مناسب کمک می‌کند. توابع عضو **setGrossSales** (تعریف شده در خطوط

57-60 از شکل ۵-۱۲) و **setCommissonRate** (تعریف شده در خطوط 72-69 از شکل ۵-۱۲)، قبل از اینکه مبادرت به تخصیص مقادیر به اعضای داده **grossSales** و **commissionRate** کنند، اعتبارسنجی آرگومان را انجام می‌دهند.

تعریف سازنده **CommissionEmployee** عمداً از گرامر مقداردهی کننده اولیه عضو در چند مثال اول این بخش استفاده نکرده است، از اینروست که می‌توانیم توضیح دهیم که چگونه تصریح کننده‌های **private** و **protected** در دسترسی به اعضا در کلاس‌های مشتق شده تاثیر می‌گذارند. همانطوری که در شکل ۵-۱۲، خطوط 15-13 مشاهده می‌کنید، اقدام به تخصیص مقادیری به اعضای داده **firstName**، **lastName** و **socialSecurityNumber** در بدنه سازنده کرده‌ایم. در انتهای این بخش به سراغ استفاده از لیست‌های مقداردهی کننده اولیه در سازنده‌ها خواهیم رفت.

```
1  // Fig. 12.4: CommissionEmployee.h
2  // CommissionEmployee class definition represents a commission employee.
3  #ifndef COMMISSION_H
4  #define COMMISSION_H
5
6  #include <string> // C++ standard string class
7  using std::string;
8
9  class CommissionEmployee
10 {
11 public:
12    CommissionEmployee( const string &, const string &, const string &,
13       double = 0.0, double = 0.0 );
14
15    void setFirstName( const string & ); // set first name
16    string getFirstName() const; // return first name
17
18    void setLastName( const string & ); // set last name
19    string getLastName() const; // return last name
20
21    void setSocialSecurityNumber( const string & ); // set SSN
22    string getSocialSecurityNumber() const; // return SSN
23
24    void setGrossSales( double ); // set gross sales amount
25    double getGrossSales() const; // return gross sales amount
26
27    void setCommissionRate( double ); // set commission rate (percentage)
28    double getCommissionRate() const; // return commission rate
29
30    double earnings() const; // calculate earnings
31    void print() const; // print CommissionEmployee object
32 private:
33    string firstName;
34    string lastName;
35    string socialSecurityNumber;
36    double grossSales; // gross weekly sales
37    double commissionRate; // commission percentage
38 }; // end class CommissionEmployee
39
40 #endif
```

شکل ٤-١٢ | فایل سرآیند کلاس CommssionEmployee.

```
1  // Fig. 12.5: CommissionEmployee.cpp
2  // Class CommissionEmployee member-function definitions.
3  #include <iostream>
```

```
4  using std::cout;
5
6  #include "CommissionEmployee.h" // CommissionEmployee class definition
7
8  // constructor
9  CommissionEmployee::CommissionEmployee(
10    const string &first, const string &last, const string &ssn,
11    double sales, double rate )
12 {
13    firstName = first; // should validate
14    lastName = last; // should validate
15    socialSecurityNumber = ssn; // should validate
16    setGrossSales( sales ); // validate and store gross sales
17    setCommissionRate( rate ); // validate and store commission rate
18 } // end CommissionEmployee constructor
19
20 // set first name
21 void CommissionEmployee::setFirstName( const string &first )
22 {
23    firstName = first; // should validate
24 } // end function setFirstName
25
26 // return first name
27 string CommissionEmployee::getFirstName() const
28 {
29    return firstName;
30 } // end function getFirstName
31
32 // set last name
33 void CommissionEmployee::setLastName( const string &last )
34 {
35    lastName = last; // should validate
36 } // end function setLastName
37
38 // return last name
39 string CommissionEmployee::getLastName() const
40 {
41    return lastName;
42 } // end function getLastName
43
44 // set social security number
45 void CommissionEmployee::setSocialSecurityNumber( const string &ssn )
46 {
47    socialSecurityNumber = ssn; // should validate
48 } // end function setSocialSecurityNumber
49
50 // return social security number
51 string CommissionEmployee::getSocialSecurityNumber() const
52 {
53    return socialSecurityNumber;
54 } // end function getSocialSecurityNumber
55
56 // set gross sales amount
57 void CommissionEmployee::setGrossSales( double sales )
58 {
59    grossSales = ( sales < 0.0 ) ? 0.0 : sales;
60 } // end function setGrossSales
61
62 // return gross sales amount
63 double CommissionEmployee::getGrossSales() const
64 {
65    return grossSales;
66 } // end function getGrossSales
67
68 // set commission rate
69 void CommissionEmployee::setCommissionRate( double rate )
70 {
71    commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
72 } // end function setCommissionRate
73
```

```
74 // return commission rate
75 double CommissionEmployee::getCommissionRate() const
76 {
77    return commissionRate;
78 } // end function getCommissionRate
79
80 // calculate earnings
81 double CommissionEmployee::earnings() const
82 {
83    return commissionRate * grossSales;
84 } // end function earnings
85
86 // print CommissionEmployee object
87 void CommissionEmployee::print() const
88 {
89    cout << "commission employee: " << firstName << ' ' << lastName
90       << "\nsocial security number: " << socialSecurityNumber
91       << "\ngross sales: " << grossSales
92       << "\ncommission rate: " << commissionRate;
93 } // end function print
```

شکل ۵-۱۲ | پیاده‌سازی فایل کلاس CommissionEmployee که نشاندهنده کارمندی است که از در صد میزان فروش حقوق دریافت می‌کند.

دقت کنید که عملیات اعتبارسنجی بر روی مقادیر آرگومان‌های سازنده یعنی **first**، **last** و **ssn** را قبل از تخصیص آنها به اعضای داده متناظر انجام نداده‌ایم. در حالیکه باید این اعتبارسنجی بر روی مقادیر صورت گیرد تا مطمئن گردیم که مقادیر در محدوده تعیین شده قرار دارند و فرمت مورد نیاز برنامه را تامین می‌کنند. مثلاً شماره تامین اجتماعی می‌بایستی نه رقم با خط تیره یا بدون خط تیره باشد (مثلاً 123456789 یا 123-45-6789).

تابع عضو **earings** (خطوط 84-81) مبادرت به محاسبه درآمد یک کارمند **CommssionEmployee** می‌کند. خط 83 مقدار **commissionRate** را در **grossSales** ضرب کرده و نتیجه را برگشت می‌دهد. تابع عضو **print** (خطوط 93-87) مقادیر کلیه عضوهای داده **CommssionEmployee** را چاپ می‌کند.

شکل ۶-۱۲ مبادرت به تست کلاس **CommssionEmployee** می‌کند. در خطوط 17-16 شی **employee** از کلاس **CommssionEmployee** ایجاد شده و سازنده برای مقداردهی اولیه شی با "Sue" بعنوان نام، "Janes" بعنوان نام خانوادگی، "222-22-222" بعنوان شماره تامین اجتماعی، 10000 بعنوان میزان فروش ناخالص و 06. بعنوان نرخ کمیسیون، فراخوانی می‌شود. خطوط 32-31 از توابع *get* برای نمایش مقادیر در این اعضای داده استفاده می‌کنند. خطوط 32-31 توابع عضو **setGrossSales** و **setCommssionRate** را برای تغییر در مقادیر اعضای داده **grossSales** و **commssionRate** فراخوانی می‌کند. سپس خط 36 تابع عضو **print** را برای نمایش اطلاعات تغییر یافته و به روز شده **CommssionEmployee** فراخوانی می‌کند. در پایان، خط 39 دستمزد محاسبه شده توسط تابع عضو

earings را با استفاده از مقادیر به روز شده اعضای داده **grossSales** و **commissionRate** بنمایش در می‌آورد.

```
1  // Fig. 12.6: fig12_06.cpp
2  // Testing class CommissionEmployee.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6  using std::fixed;
7
8  #include <iomanip>
9  using std::setprecision;
10
11 #include "CommissionEmployee.h" // CommissionEmployee class definition
12
13 int main()
14 {
15    // instantiate a CommissionEmployee object
16    CommissionEmployee employee(
17       "Sue", "Jones", "222-22-2222", 10000, .06 );
18
19    // set floating-point output formatting
20    cout << fixed << setprecision( 2 );
21
22    // get commission employee data
23    cout << "Employee information obtained by get functions: \n"
24       << "\nFirst name is " << employee.getFirstName()
25       << "\nLast name is " << employee.getLastName()
26       << "\nSocial security number is "
27       << employee.getSocialSecurityNumber()
28       << "\nGross sales is " << employee.getGrossSales()
29      << "\nCommission rate is " << employee.getCommissionRate() << endl;
30
31    employee.setGrossSales( 8000 ); // set gross sales
32    employee.setCommissionRate( .1 ); // set commission rate
33
34    cout << "\nUpdated employee information output by print function: \n"
35       << endl;
36    employee.print(); // display the new employee information
37
38    // display the employee's earnings
39    cout << "\n\nEmployee's earnings: $" << employee.earnings() << endl;
40
41    return 0;
42 } // end main
```

```
Employee information obtained by get functions:

First name is Sue
Last name is Jones
Social security number is 222-22-2222
Gross sales is 10000.00
Commission rate is 0.06

Update employee information output by print function:

commission employee: Sue Jones
social security number: 222-22-2222
commission rate: 0.10

Employee's earnings: $800.00
```

شکل ٦-١٢ | برنامه تست کننده کلاس CommssionEmployee.

٢-٤-١٢ ایجاد کلاس BasePlusCommssionEmployee بعنوان ارث‌بری

در این بخش به سراغ قسمت دوم از مقدمه و معرفی ارث‌بری می‌رویم و آنرا با ایجاد دو تست کلاس **BasePlusCommssionEmployee** (شکل‌های ۷-۱۲ و ۸-۱۲) انجام می‌دهیم که حاوی نام، نام خانوادگی، شماره تامین اجتماعی، میزان فروش ناخالص، نرخ کمیسیون و حقوق پایه است. (این کلاس را بصورت مستقل و کاملاً جدید ایجاد می‌کنیم).

تعریف کلاس BasePlusCommissionEmploye

فایل سرآیند **BasePlusCommssionEmployee** در شکل ۷-۱۲ تصریح کننده سرویس‌های سراسری (public) کلاس است که شامل سازنده این کلاس (خطوط 14-13) و توابع عضو **earings** (خط 34) و **print** (خط 35) است.

```
1  // Fig. 12.7: BasePlusCommissionEmployee.h
2  // BasePlusCommissionEmployee class definition represents an employee
3  // that receives a base salary in addition to commission.
4  #ifndef BASEPLUS_H
5  #define BASEPLUS_H
6
7  #include <string> // C++ standard string class
8  using std::string;
9
10 class BasePlusCommissionEmployee
11 {
12 public:
13    BasePlusCommissionEmployee( const string &, const string &,
14       const string &, double = 0.0, double = 0.0, double = 0.0 );
15
16    void setFirstName( const string & ); // set first name
17    string getFirstName() const; // return first name
18
19    void setLastName( const string & ); // set last name
20    string getLastName() const; // return last name
21
22    void setSocialSecurityNumber( const string & ); // set SSN
23    string getSocialSecurityNumber() const; // return SSN
24
25    void setGrossSales( double ); // set gross sales amount
26    double getGrossSales() const; // return gross sales amount
27
28    void setCommissionRate( double ); // set commission rate
29    double getCommissionRate() const; // return commission rate
30
31    void setBaseSalary( double ); // set base salary
32    double getBaseSalary() const; // return base salary
33
34    double earnings() const; // calculate earnings
35    void print() const; // print BasePlusCommissionEmployee object
36 private:
37    string firstName;
38    string lastName;
39    string socialSecurityNumber;
40    double grossSales; // gross weekly sales
41    double commissionRate; // commission percentage
42    double baseSalary; // base salary
43 }; // end class BasePlusCommissionEmployee
44
45 #endif
```

شکل ۷-۱۲ | فایل سرآیند کلاس BasePlusCommssionEmployee.

خطوط 32-16 توابع سراسری *get* و *set* را برای اعضای داده خصوصی کلاس (اعلان شده در خطوط 42-37) بنام‌های **firstName**، **lastName**، **socialSecurityNumber**، **grossSales** (فروش ناخالص)، **commissionRate** و (نرخ کمیسیون) **baseSalary** (حقوق پایه) اعلان کرده‌اند. این متغیرها و توابع عضو تمام ویژگیهای ضروری یک کارمند که دارای حقوق پایه و کمیسیون دریافتی است را کپسوله می‌کند. به شباهت موجود مابین این کلاس و کلاس **CommssionEmployee** (شکل‌های ۴-۱۲ و ۵-۱۲) توجه کنید. در این مثال، هنوز قصد توضیح شباهت‌ها را نداریم.

تابع عضو **earnings** (تعریف شده در خطوط 99-96 از شکل ۸-۱۲) مبادرت محاسبه حقوق این نوع کارمند می‌کند. خط 98 نتیجه افزودن حقوق پایه کارمند به حاصلضرب نرخ کمیسیون و فروش ناخالص را برگشت می‌دهد.

تست کلاس BasePlusCommssionEmployee

شکل ۹-۱۲ تست کننده کلاس **BasePlusCommssionEmployee** است. خطوط 18-17 مبادرت به ایجاد یک شی **Employee** از این کلاس کرده و "Bob" و "Lewis"، "333-33-3333"، "5000"، ".04" و 300 را بترتیب بعنوان نام، نام خانوادگی، شماره تامین اجتماعی، فروش ناخالص، نرخ کمیسیون و حقوق پایه به سازنده ارسال می‌کنند. خطوط 31-24 از توابع *get* این کلاس برای بازیابی مقادیر اعضای داده شی در خروجی استفاده می‌کنند. خط 33 تابع عضو **setBaseSalary** را برای تغییر دادن حقوق پایه احضار می‌کند.

تابع عضو **setBaseSalary** (شکل ۸-۱۲، خطوط 87-84) ما را مطمئن می‌سازد که داده عضو **baseSalary** (حقوق پایه) هرگز یک مقدار منفی نباشد، چرا که حقوق پایه یک کارمند نمی‌تواند منفی باشد. خط 37 از شکل ۹-۱۲ تابع عضو **print** را برای چاپ (نمایش) اطلاعات به روز شده کلاس و خط 40 تابع عضو **earnings** را برای نمایش حقوق کارمند فراخوانی می‌کند.

```
1  // Fig. 12.8: BasePlusCommissionEmployee.cpp
2  // Class BasePlusCommissionEmployee member-function definitions.
3  #include <iostream>
4  using std::cout;
5
6  // BasePlusCommissionEmployee class definition
7  #include "BasePlusCommissionEmployee.h"
8
9  // constructor
10 BasePlusCommissionEmployee::BasePlusCommissionEmployee(
11    const string &first, const string &last, const string &ssn,
12    double sales, double rate, double salary )
13 {
14    firstName = first; // should validate
15    lastName = last; // should validate
16    socialSecurityNumber = ssn; // should validate
```

```
17    setGrossSales( sales ); // validate and store gross sales
18    setCommissionRate( rate ); // validate and store commission rate
19    setBaseSalary( salary ); // validate and store base salary
20 } // end BasePlusCommissionEmployee constructor
21
22 // set first name
23 void BasePlusCommissionEmployee::setFirstName( const string &first )
24 {
25    firstName = first; // should validate
26 } // end function setFirstName
27
28 // return first name
29 string BasePlusCommissionEmployee::getFirstName() const
30 {
31    return firstName;
32 } // end function getFirstName
33
34 // set last name
35 void BasePlusCommissionEmployee::setLastName( const string &last )
36 {
37    lastName = last; // should validate
38 } // end function setLastName
39
40 // return last name
41 string BasePlusCommissionEmployee::getLastName() const
42 {
43    return lastName;
44 } // end function getLastName
45
46 // set social security number
47 void BasePlusCommissionEmployee::setSocialSecurityNumber(
48    const string &ssn )
49 {
50    socialSecurityNumber = ssn; // should validate
51 } // end function setSocialSecurityNumber
52
53 // return social security number
54 string BasePlusCommissionEmployee::getSocialSecurityNumber() const
55 {
56    return socialSecurityNumber;
57 } // end function getSocialSecurityNumber
58
59 // set gross sales amount
60 void BasePlusCommissionEmployee::setGrossSales( double sales )
61 {
62    grossSales = ( sales < 0.0 ) ? 0.0 : sales;
63 } // end function setGrossSales
64
65 // return gross sales amount
66 double BasePlusCommissionEmployee::getGrossSales() const
67 {
68    return grossSales;
69 } // end function getGrossSales
70
71 // set commission rate
72 void BasePlusCommissionEmployee::setCommissionRate( double rate )
73 {
74    commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
75 } // end function setCommissionRate
76
77 // return commission rate
78 double BasePlusCommissionEmployee::getCommissionRate() const
79 {
80    return commissionRate;
81 } // end function getCommissionRate
82
83 // set base salary
84 void BasePlusCommissionEmployee::setBaseSalary( double salary )
85 {
86    baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
```

```
87 } // end function setBaseSalary
88
89 // return base salary
90 double BasePlusCommissionEmployee::getBaseSalary() const
91 {
92    return baseSalary;
93 } // end function getBaseSalary
94
95 // calculate earnings
96 double BasePlusCommissionEmployee::earnings() const
97 {
98    return baseSalary + ( commissionRate * grossSales );
99 } // end function earnings
100
101 // print BasePlusCommissionEmployee object
102 void BasePlusCommissionEmployee::print() const
103 {
104    cout << "base-salaried commission employee: " << firstName << ' '
105       << lastName << "\nsocial security number: " << socialSecurityNumber
106       << "\ngross sales: " << grossSales
107       << "\ncommission rate: " << commissionRate
108       << "\nbase salary: " << baseSalary;
109 } // end function print
```

شکل ۸-۱۲ | کلاس BasePlusCommssionEmployee نشاندهنده کارمندی است که علاوه بر حقوق پایه، کمیسیونی هم دریافت می‌کند.

```
1  // Fig. 12.9: fig12_09.cpp
2  // Testing class BasePlusCommissionEmployee.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6  using std::fixed;
7
8  #include <iomanip>
9  using std::setprecision;
10
11 // BasePlusCommissionEmployee class definition
12 #include "BasePlusCommissionEmployee.h"
13
14 int main()
15 {
16    // instantiate BasePlusCommissionEmployee object
17    BasePlusCommissionEmployee
18       employee( "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
19
20    // set floating-point output formatting
21    cout << fixed << setprecision( 2 );
22
23    // get commission employee data
24    cout << "Employee information obtained by get functions: \n"
25       << "\nFirst name is " << employee.getFirstName()
26       << "\nLast name is " << employee.getLastName()
27       << "\nSocial security number is "
28       << employee.getSocialSecurityNumber()
29       << "\nGross sales is " << employee.getGrossSales()
30       << "\nCommission rate is " << employee.getCommissionRate()
31       << "\nBase salary is " << employee.getBaseSalary() << endl;
32
33    employee.setBaseSalary( 1000 ); // set base salary
34
35    cout << "\nUpdated employee information output by print function: \n"
36       << endl;
37    employee.print(); // display the new employee information
38
39    // display the employee's earnings
40    cout << "\n\nEmployee's earnings: $" << employee.earnings() << endl;
41
42    return 0;
```

```
43 } // end main
```

```
Employee information obtained by get functions:

First name is Bob
Last name is Lewis
Social security number is 333-33-3333
Gross sales is 5000.00
Commission rate is 0.04
Base salary is 300.00

Update employee information output by print function:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 1000.00

Employee's earnings: $1200.00
```

شکل ۹-۱۲ | برنامه تست کننده کلاس BasePlusCommssionEmployee.

بررسی شباهت‌های مابین کلاس BasePlusCommssionEmployee و کلاس CommssionEmployee

به میزان کد بکار رفته برای کلاس **BasePlusCommssionEmployee** (شکل‌های ۷-۱۲ و ۸-۱۲) دقت کنید که تقریباً برابر با کد بکار رفته برای کلاس **CommssionEmployee** است (شکل‌های ۴-۱۲ و ۵-۱۲) برای مثال، در کلاس **BasePlusCommssionEmployee** اعضای داده خصوصی عبارتند از **firstName** و **lastName** و توابع عضو **setFirstName**، **getFirstName**، **setLastName** و **getLastName** که یکسان با موارد مشابه در کلاس **CommssionEmployee** هستند. همچنین هر دو کلاس **CommssionEmployee** و **BasePlusCommssionEmployee** حاوی اعضای داده خصوصی **socialSecurityNumber**، **commissionRate** و **grossSales** به همراه توابع *get* و *set* بمنظور کار با این اعضا هستند. علاوه بر این سازنده **BasePlusCommssionEmpolyee** تقریباً یکسان با سازنده کلاس **BasePlusCommssionEmployee** است،بجزاینکه سازنده **BasePlusCommssionEmployee** مبادرت به تست **baseSalary** هم می‌کند. موارد دیگر در کلاس **BasePlusCommssionEmployee** عبارتند از اعضای داده خصوصی **baseSalary** و توابع عضو **setBaseSalary** و **getBaseSalary**. تابع **print** این کلاس شبیه تابع **print** موجود در کلاس **CommissionEmployee** است، بجز اینکه تابع **print** در **BasePlusCommssionEmployee** مقدار عضو داده **baseSalary** را هم چاپ می‌کند.

می‌توانیم کلاس **BasePlusCommssionEmployee** را با کپی کدها از کلاس **CommissionEmplyee** و سپس اصلاح کلاس **BasePlusCommssionEmployee** برای در برداشتن یک حقوق پایه و توابع عضو که برای کار با حقوق پایه لازم هستند، ایجاد کنیم. غالباً این روش کپی کردن، زمینه‌ساز خطا بوده و زمانبر است. بدتر از آن می‌تواند بصورت کپی‌های متعدد از کد یکسان در کل سیستم پخش شود، ایجاد و

نگهداری چنین کدی کابوس است. بهترین راه حل استفاده از توارث است که اعضای داده و توابع عضو از یک کلاس را بعنوان بخش‌های از کلاس‌های دیگر جذب می‌کند، بدون اینکه کد تکثیر شده باشد.

۳-٤-۱۲ ایجاد سلسله مراتب توارث BasePlusCommssionEmployee

در این بخش یک نسخه جدید از کلاس **BasePlusCommssionEmployee** ایجاد و تست می‌کنیم (شکل‌های ۱۰-۱۲ و ۱۱-۱۲) که از کلاس **CommissionEmployee** (شکل‌های ۴-۱۲ و ۵-۱۲) مشتق شده است. در این مثال شی **BasePlusCommssionEmployee** یک، **CommisionEmploye** است (چرا که قابلیت‌های کلاس **CommissionEmployee** ارث‌برده می‌شود)، اما کلاس **BasePlusCommssionEmployee** دارای عضو داده **baseSalary** متعلق ب‌خود است (شکل ۱۰-۱۲، خط 20).

```
1  // Fig. 12.10: BasePlusCommissionEmployee.h
2  // BasePlusCommissionEmployee class derived from class
3  // CommissionEmployee.
4  #ifndef BASEPLUS_H
5  #define BASEPLUS_H
6
7  #include <string> // C++ standard string class
8  using std::string;
9
10 #include "CommissionEmployee.h" // CommissionEmployee class declaration
11
12 class BasePlusCommissionEmployee : public CommissionEmployee
13 {
14 public:
15    BasePlusCommissionEmployee( const string &, const string &,
16       const string &, double = 0.0, double = 0.0, double = 0.0 );
17
18    void setBaseSalary( double ); // set base salary
19    double getBaseSalary() const; // return base salary
20
21    double earnings() const; // calculate earnings
22    void print() const; // print BasePlusCommissionEmployee object
23 private:
24    double baseSalary; // base salary
25 }; // end class BasePlusCommissionEmployee
26
27 #endif
```

شکل ۱۰-۱۲ | تعریف کلاس BasePlusCommssionEmployee شامل رابطه توارث از کلاس CommissionEmployee.

نماد کولن (:) در خط 12 از تعریف کلاس بر این نکته دلالت دارد که کلاس حالت ارث‌بری دارد. کلمه‌کلیدی **public** نشاندهنده نوع توارث است. بعنوان یک کلاس مشتق شده (شکل یافته با توارث **public**)، **BasePlusCommssionEmployee** تمام اعضای کلاس **CommissionEmployee** را بجز سازنده به ارث می‌برد [توجه کنید که نابود کننده‌ها به ارث برده نمی‌شوند]. از اینرو، سرویس‌های سراسری **BasePlusCommssionEmployee** شامل سازنده خود بوده (خطوط 16-15) و توابع عضو

سراسری از کلاس **CommissionEmloyee** به ارث برده می‌شوند. اگرچه نمی‌توانیم در کد منبع **BasePlusCommssionEmployee** این توابع به ارث رفته را مشاهده کنیم، با اینحال آنها بخشی از کلاس مشتق شده **BasePlusCommssionEmployee** هستند. همچنین سرویس‌های سراسری کلاس مشتق شده شامل توابع عضو **setBaseSalary**، **getBaseSalary**، **earnngs** و **print** می‌باشند (خطوط 18-22).

در شکل ۱۱-۱۲ پیاده‌سازی تابع عضو متعلق **BasePlusCommssionEmployee** نشان داده شده است. سازنده در خطوط 17-10 به معرفی گرامر مقداردهی کننده اولیه کلاس مبنا (خط 14) پرداخته است که از یک مقداردهی کننده اولیه عضو برای ارسال آرگومان‌ها به سازنده کلاس مبنا استفاده می‌کند.

برای فراخوانی سازنده کلاس مبنا به منظور مقداردهی اولیه، اعضای داده کلاس مبنا که توسط کلاس مشتق شده به ارث برده می‌شوند، C++ نیازمند یک سازنده کلاس مشتق شده است. خط 14 این وظیفه را با احضار سازنده **CommissionEmployee** با نام، ارسال پارامترهای سازنده **first**، **last**، **ssn**، **sales** و **rate** بعنوان آرگومان‌های برای مقداردهی اولیه اعضای داده کلاس **firstName**، **lastName**، **socialSecurityNumber**، **grossSales** و **CommissinRate** بکار گرفته می‌شوند، انجام می‌دهد. اگر سازنده **BasePlusCommssionEmployee** نتواند صریحاً سازنده کلاس **CommssionEmployee** را فراخوانی کند. **C++** مبادرت به احضار سازنده پیش‌فرض **CommissionEmploye** خواهد کرد، اما این کلاس فاقد چنین سازنده‌ای است و از اینرو کامپایلر یک خطا صادر می‌کند. از فصل سوم بخاطر دارید که کامپایلر یک سازنده پیش‌فرض بدون پارامتر برای هر کلاسی که بطور صریح سازنده‌ای را در نظر نگرفته است، فراخوانی می‌کند. با این وجود، **CommissionEmploye** بطور صریح دارای یک سازنده بوده و نیازی به سازنده پیش‌فرض نیست و هر عملی که بخواهد سازنده پیش‌فرض را برای این کلاس فراخوانی کند با خطای کامپایل مواجه می‌شود.

```
1  // Fig. 12.11: BasePlusCommissionEmployee.cpp
2  // Class BasePlusCommissionEmployee member-function definitions.
3  #include <iostream>
4  using std::cout;
5
6  // BasePlusCommissionEmployee class definition
7  #include "BasePlusCommissionEmployee.h"
8
9  // constructor
10 BasePlusCommissionEmployee::BasePlusCommissionEmployee(
11    const string &first, const string &last, const string &ssn,
12    double sales, double rate, double salary )
13    // explicitly call base-class constructor
14    : CommissionEmployee( first, last, ssn, sales, rate )
15 {
16    setBaseSalary( salary ); // validate and store base salary
17 } // end BasePlusCommissionEmployee constructor
18
```

```
19 // set base salary
20 void BasePlusCommissionEmployee::setBaseSalary( double salary )
21 {
22    baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
23 } // end function setBaseSalary
24
25 // return base salary
26 double BasePlusCommissionEmployee::getBaseSalary() const
27 {
28    return baseSalary;
29 } // end function getBaseSalary
30
31 // calculate earnings
32 double BasePlusCommissionEmployee::earnings() const
33 {
34    // derived class cannot access the base class's private data
35    return baseSalary + ( commissionRate * grossSales );
36 } // end function earnings
37
38 // print BasePlusCommissionEmployee object
39 void BasePlusCommissionEmployee::print() const
40 {
41    // derived class cannot access the base class's private data
42   cout << "base-salaried commission employee: " << firstName << ' '
43       << lastName << "\nsocial security number: " << socialSecurityNumber
44       << "\ngross sales: " << grossSales
45       << "\ncommission rate: " << commissionRate
46       << "\nbase salary: " << baseSalary;
47 } // end function print
```

```
C:\cpphtp5_examples\ch12\Fig12_10_11\BasePlusCommission-Employee.cpp(35):
   error C2248:'CommissionEmployee::commissionRate':
   cannot access private member declared in class 'CommissionEmployee'
   C:\cpphtp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(37):
   see declaration of 'CommissionEmployee::commissionRate'
   C:\cpphtp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(10):
   see declaration of 'CommissionEmployee'

C:\cpphtp5_examples\ch12\Fig12_10_11\BasePlusCommission-Employee.cpp(35):
   error C2248:'CommissionEmployee::grossSales':
   cannot access private member declared in class 'CommissionEmployee'
   C:\cpphtp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(36):
   see declaration of 'CommissionEmployee::grossSales'
   C:\cpphtp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(10):
   see declaration of 'CommissionEmployee'

C:\cpphtp5_examples\ch12\Fig12_10_11\BasePlusCommission-Employee.cpp(42):
   error C2248:'CommissionEmployee::firstName':
   cannot access private member declared in class 'CommissionEmployee'
   C:\cpphtp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(33):
   see declaration of 'CommissionEmployee::firstName'
   C:\cpphtp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(10):
   see declaration of 'CommissionEmployee'

C:\cpphtp5_examples\ch12\Fig12_10_11\BasePlusCommission-Employee.cpp(43):
   error C2248:'CommissionEmployee::lastName':
   cannot access private member declared in class 'CommissionEmployee'
   C:\cpphtp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(34):
   see declaration of 'CommissionEmployee::lastName'
   C:\cpphtp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(10):
   see declaration of 'CommissionEmployee'

C:\cpphtp5_examples\ch12\Fig12_10_11\BasePlusCommission-Employee.cpp(43):
   error C2248:'CommissionEmployee::socialSecurity-Number':
   cannot access private member declared in class 'CommissionEmployee'
   C:\cpphtp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(35):
   see declaration of 'CommissionEmployee::socialSecurityNumber'
   C:\cpphtp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(10):
   see declaration of 'CommissionEmployee'

C:\cpphtp5_examples\ch12\Fig12_10_11\BasePlusCommission-Employee.cpp(44):
```

```
    error C2248:'CommissionEmployee::grossSales':
    cannot access private member declared in class 'CommissionEmployee'
    C:\cpphtp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(36):
    see declaration of 'CommissionEmployee::grossSales'
    C:\cpphtp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(10):
    see declaration of 'CommissionEmployee'

C:\cpphtp5_examples\ch12\Fig12_10_11\BasePlusCommission-Employee.cpp(45):
    error C2248:'CommissionEmployee::commissionRate':
    cannot access private member declared in class 'CommissionEmployee'
    C:\cpphtp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(37):
    see declaration of 'CommissionEmployee::commissionRate'
    C:\cpphtp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(10):
    see declaration of 'CommissionEmployee'
```

شکل ۱۱-۱۲ | فایل پیاده‌سازی BasePlusCommssionEmployee: داده خصوصی کلاس مبنا از طریق کلاس مشتق شده در دسترس نمی‌باشد.

کامپایلر خطای برای خط 35 از شکل ۱۱-۱۲ تولید می‌کند، چرا که اعضای داده کلاس مبنا **CommissionEmployee** بنام **commisionRate** و **grossSales** خصوصی (private) هستند و توابع عضو کلاس مشتق شده اجازه دسترسی به داده خصوصی کلاس مبنا را ندارند. کامپایلر چندین پیغام خطای دیگر را در خطوط 45-42 بر روی تابع عضو **print** به همین دلیل صادر می‌کند. همانطوری که مشاهده می‌کنید C++ در خصوص دسترسی به اعضای داده بسیار سختگیر است، از اینرو حتی یک کلاس مشتق شده (که عاقبت مرتبط با کلاس مبنای خود است) نمی‌تواند به داده خصوصی کلاس مبنا دسترسی داشته باشد.

ما عمداً این کد اشتباه را وارد برنامه شکل ۱۱-۱۲ کرده‌ایم تا نشان دهیم که توابع عضو یک کلاس مشتق شده نمی‌توانند به داده خصوصی کلاس مبنای خود دسترسی پیدا کنند. می‌توان با استفاده از توابع *get* که از کلاس **CommissionEmployee** ارث‌بری می‌شوند جلوی این خطاها را گرفت. برای مثال، خط 35 می‌تواند **getCommissionRate** و **getGrossRate** را برای دسترسی به داده خصوصی **commissionRate** و **grossSales** کلاس **CommissionEmployee** فراخوانی کند. به همین ترتیب، خطوط 45-42 می‌توانند از توابع *get* مناسب برای بازیابی مقادیر از اعضای داده کلاس مبنا استفاده کنند. در مثال بعدی، نحوه استفاده از داده **protected** را نشان خواهیم داد که امکان می‌دهد تا از خطای رخ داده در این مثال جلوگیری کنیم.

وارد ساختن فایل سرآیند کلاس مبنا در فایل سرآیند کلاس مشتق شده با #include

توجه کنید که **#include** فایل سرآیند کلاس مبنا را در فایل سرآیند کلاس مشتق شده قرار داده‌ایم (خط 10 از شکل ۱۰-۱۲). انجام اینکار به سه دلیل ضروری است. اول اینکه، کلاس مشتق شده برای

استفاده از نام کلاس در خط 12 نیاز دارد تا به کامپایلر اعلان کند که کلاس مبنا موجود است. تعریف کلاس دقیقاً در **CommissionEmpolyee.h** است.

دلیل دوم این است که کامپایلر از تعریف کلاس برای تعیین سایز شی از آن کلاس استفاده می‌کند (در بخش ۸-۳ در این مورد صحبت کرده‌ایم). یک برنامه سرویس‌گیرنده که یک شی از کلاس ایجاد می‌کند بایستی تعریف کلاس را **#include** نماید تا کامپایلر بتواند به میزان مناسب برای آن شی حافظه رزرو نماید.

به هنگام استفاده از توارث، سایز یک شی از کلاس مشتق شده بستگی به اعضای داده اعلان شده در تعریف کلاس داشته و اعضای داده آنرا مستقیماً و غیرمستقیم از کلاس مبنا به ارث می‌برند. با وارد کردن تعریف کلاس در خط 10 به کامپایلر اجازه داده می‌شود تا حافظه مورد نیاز برای اعضای داده کلاس مبنا که بخشی از شی از کلاس مشتق شده می‌باشند تامین شده و از اینرو کل سایز تخصیصی شامل این موارد نیز می‌شود.

دلیل آخر برای خط 10 امکان دادن به کامپایلر برای تعیین اینکه آیا کلاس مشتق شده از اعضای به ارث برده شده کلاس مبنا بدرستی استفاده می‌کند یا خیر. برای مثال در برنامه شکل‌های ۱۰-۱۲ و ۱۱-۱۲، کامپایلر از فایل سرآیند کلاس مبنا برای تعیین اینکه اعضای داده در دسترس کلاس مشتق شده از نوع **private** در کلاس مبنا هستند یا خیر، استفاده کرده است. از آنجا که این نوع داده‌ها در دسترس کلاس مشتق شده قرار داده نمی‌شوند، کامپایلر خطا تولید می کند.

فرآیند لینک در سلسله مراتب توارث

در بخش ۹-۳، در ارتباط با فرآیند لینک در ایجاد یک برنامه کاربردی بنام **GradeBook** صحبت کردیم. در آن مثال، مشاهده کردید که شی سرویس‌گیرنده با کد شی کلاس **GradeBook** به همراه هر کلاس بکار رفته از کتابخانه استاندارد C++ لینک شد.

فرآیند لینک در برنامه‌ای که از کلاس‌های به ارث رفته استفاده می‌کند، مشابه است. فرآیند مستلزم کد شی برای تمام کلاس‌های بکار رفته در برنامه و کد شی بکار رفته چه بصورت مستقیم و غیرمستقیم از کلاس‌های مبنا در هر کلاس مشتق شده‌ای در برنامه است. فرض کنید سرویس گیرنده‌ای می‌خواهد برنامه‌ای ایجاد کند که از کلاس **BasePlusCommssionEmployee** استفاده کند که خود از کلاس **CommissionEmployee** مشتق شده است. در زمان کامپایل برنامه سرویس‌گیرنده کد شی سرویس‌گیرنده بایستی با کد شی کلاس‌های **BasePlusCommssionEmployee** و

CommissionEmployee لینک شده باشد، چرا که **BasePlusCommssionEmployee** توابع عضو را از کلاس مبنا **CommissionEmployee** ارث می‌برد. همچنین کد با کد شی هر کلاسی از کتابخانه استاندارد C++ که در کلاس **CommssionEmployee** و کلاس **BasePlusCommssionEmployee** یا کد سرویس‌گیرنده بکار رفته لینک می‌شود. در اینحالت برنامه قادر به دسترسی به پیاده‌سازی تمام توابع در برنامه خواهد بود.

۱۲-٤-٤ ایجاد سلسله مراتب توارث CommssionEmployee-BasePlusCommssionEmployee با استفاده از داده protected

برای اینکه کلاس **BasePlusCommssionEmployee** بتواند بطور مستقیم به اعضای داده **firstName**، **lastName**، **socialSecurityNumber**، **grossSales** و **commssionRate** از کلاس **CommissionEmployee** دسترسی داشته باشد، می‌توانیم این اعضا را بصورت **protected** (حفاظت شده) در کلاس مبنا اعلان کنیم. همانطوری که در بخش ۲-۱۲ توضیح داده شد، اعضای **protected** کلاس مبنا می‌توانند توسط اعضا و دوستان (friend) کلاس مبنا و اعضا و دوستان هر کلاس مشتق شده از آن کلاس مبنا در دسترس قرار گیرند.

تعریف کلاس مبنا CommissionEmployee با داده protected

کلاس **CommiesionEmployee** در برنامه شکل‌های ۱۲-۱۲ و ۱۳-۱۲ مبادرت به اعلان اعضای داده **firstName**، **lastName**، **socialSecurityNumber**، **grossSales** و **commissionRate** بصورت **protected** (شکل ۱۲-۱۲، خطوط 37-33) بجای **private** کرده است. پیاده‌سازی تابع عضو در شکل ۱۳-۱۲ همانند شکل ۵-۱۲ است.

```
1  // Fig. 12.12: CommissionEmployee.h
2  // CommissionEmployee class definition with protected data.
3  #ifndef COMMISSION_H
4  #define COMMISSION_H
5
6  #include <string> // C++ standard string class
7  using std::string;
8
9  class CommissionEmployee
10 {
11 public:
12    CommissionEmployee( const string &, const string &, const string &,
13       double = 0.0, double = 0.0 );
14
15    void setFirstName( const string & ); // set first name
16    string getFirstName() const; // return first name
17
18    void setLastName( const string & ); // set last name
19    string getLastName() const; // return last name
20
21    void setSocialSecurityNumber( const string & ); // set SSN
22    string getSocialSecurityNumber() const; // return SSN
```

```
23
24    void setGrossSales( double ); // set gross sales amount
25    double getGrossSales() const; // return gross sales amount
26
27    void setCommissionRate( double ); // set commission rate
28    double getCommissionRate() const; // return commission rate
29
30    double earnings() const; // calculate earnings
31    void print() const; // print CommissionEmployee object
32 protected:
33    string firstName;
34    string lastName;
35    string socialSecurityNumber;
36    double grossSales; // gross weekly sales
37    double commissionRate; // commission percentage
38 }; // end class CommissionEmployee
39
40 #endif
```

شکل ۱۲-۱۲ | تعریف کلاس CommssionEmployee که به داده protected اعلان شده اجازه دسترسی توسط کلاس‌های مشتق شده را می‌دهد.

```
1  // Fig. 12.13: CommissionEmployee.cpp
2  // Class CommissionEmployee member-function definitions.
3  #include <iostream>
4  using std::cout;
5
6  #include "CommissionEmployee.h" // CommissionEmployee class definition
7
8  // constructor
9  CommissionEmployee::CommissionEmployee(
10    const string &first, const string &last, const string &ssn,
11    double sales, double rate )
12 {
13    firstName = first; // should validate
14    lastName = last; // should validate
15    socialSecurityNumber = ssn; // should validate
16    setGrossSales( sales ); // validate and store gross sales
17    setCommissionRate( rate ); // validate and store commission rate
18 } // end CommissionEmployee constructor
19
20 // set first name
21 void CommissionEmployee::setFirstName( const string &first )
22 {
23    firstName = first; // should validate
24 } // end function setFirstName
25
26 // return first name
27 string CommissionEmployee::getFirstName() const
28 {
29    return firstName;
30 } // end function getFirstName
31
32 // set last name
33 void CommissionEmployee::setLastName( const string &last )
34 {
35    lastName = last; // should validate
36 } // end function setLastName
37
38 // return last name
39 string CommissionEmployee::getLastName() const
40 {
41    return lastName;
42 } // end function getLastName
43
44 // set social security number
45 void CommissionEmployee::setSocialSecurityNumber( const string &ssn )
46 {
47    socialSecurityNumber = ssn; // should validate
48 } // end function setSocialSecurityNumber
```

```
49
50 // return social security number
51 string CommissionEmployee::getSocialSecurityNumber() const
52 {
53    return socialSecurityNumber;
54 } // end function getSocialSecurityNumber
55
56 // set gross sales amount
57 void CommissionEmployee::setGrossSales( double sales )
58 {
59    grossSales = ( sales < 0.0 ) ? 0.0 : sales;
60 } // end function setGrossSales
61
62 // return gross sales amount
63 double CommissionEmployee::getGrossSales() const
64 {
65    return grossSales;
66 } // end function getGrossSales
67
68 // set commission rate
69 void CommissionEmployee::setCommissionRate( double rate )
70 {
71    commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
72 } // end function setCommissionRate
73
74 // return commission rate
75 double CommissionEmployee::getCommissionRate() const
76 {
77    return commissionRate;
78 } // end function getCommissionRate
79
80 // calculate earnings
81 double CommissionEmployee::earnings() const
82 {
83    return commissionRate * grossSales;
84 } // end function earnings
85
86 // print CommissionEmployee object
87 void CommissionEmployee::print() const
88 {
89    cout << "commission employee: " << firstName << ' ' << lastName
90       << "\nsocial security number: " << socialSecurityNumber
91       << "\ngross sales: " << grossSales
92       << "\ncommission rate: " << commissionRate;
93 } // end function print
```

شکل ۱۳-۱۲ | کلاس CommssionEmployee با داده protected.

اصلاح کلاس مشتق شده BasePlusCommssionEmployee

اکنون مبادرت به اصلاح کلاس **BasePlusCommssionEmployee** (شکل‌های ۱۴-۱۲ و ۱۵-۱۲) می‌کنیم تا بتواند از نسخه کلاس **CommisonEmplyee** در شکل‌های ۱۲-۱۲ و ۱۳-۱۲ ارث‌بری داشته باشد. بدلیل اینکه کلاس **BasePlusCommssionEmployee** از این نسخه از کلاس ارث‌بری دارد، شی‌های کلاس **BasePlusCommssionEmployee** می‌توانند به عضوهای داده به ارث رفته که بصورت **protected** در کلاس **BasePlusCommssionEmployee** اعلان شده‌اند، دسترسی پیدا کنند (یعنی اعضای **firstName**، **lastName**، **socialSecurityNumber**، **grossSales** و **commsionRate**). در

نتیجه، کامپایلر به هنگام کامپایل توابع عضو **earnings** و **print** که در شکل ۱۵-۱۲ تعریف شده‌اند، خطا تولید نخواهد کرد.

```
1  // Fig. 12.14: BasePlusCommissionEmployee.h
2  // BasePlusCommissionEmployee class derived from class
3  // CommissionEmployee.
4  #ifndef BASEPLUS_H
5  #define BASEPLUS_H
6
7  #include <string> // C++ standard string class
8  using std::string;
9
10 #include "CommissionEmployee.h" // CommissionEmployee class declaration
11
12 class BasePlusCommissionEmployee : public CommissionEmployee
13 {
14 public:
15    BasePlusCommissionEmployee( const string &, const string &,
16       const string &, double = 0.0, double = 0.0, double = 0.0 );
17
18    void setBaseSalary( double ); // set base salary
19    double getBaseSalary() const; // return base salary
20
21    double earnings() const; // calculate earnings
22    void print() const; // print BasePlusCommissionEmployee object
23 private:
24    double baseSalary; // base salary
25 }; // end class BasePlusCommissionEmployee
26
27 #endif
```

شکل ۱۴-۱۲ | فایل سرآیند کلاس BasePlusCommssionEmployee.

شکل ۱۵-۱۲ فایل پیاده‌سازی **BasePlusCommssionEmployee** است که داده **protected** از **CommsionEmployee** را به ارث می‌برد. کلاس **BasePlusCommssionEmployee** سازنده کلاس **CommsionEmplyee** را به ارث نمی‌برد. با این وجود سازنده کلاس **BasePlusCommssionEmployee** اقدام به فراخوانی صریح سازنده **CommssionEmployee** می‌کند (شکل ۱۵-۱۲، خطوط 17-10)، چرا که **CommssionEmployee** حاوی یک سازنده پیش‌فرض نیست که بتواند آنرا بصورت ضمنی (غیرصریح) را فراخوانی نماید.

```
1  // Fig. 12.15: BasePlusCommissionEmployee.cpp
2  // Class BasePlusCommissionEmployee member-function definitions.
3  #include <iostream>
4  using std::cout;
5
6  // BasePlusCommissionEmployee class definition
7  #include "BasePlusCommissionEmployee.h"
8
9  // constructor
10 BasePlusCommissionEmployee::BasePlusCommissionEmployee(
11    const string &first, const string &last, const string &ssn,
12    double sales, double rate, double salary )
13    // explicitly call base-class constructor
14    : CommissionEmployee( first, last, ssn, sales, rate )
15 {
16    setBaseSalary( salary ); // validate and store base salary
17 } // end BasePlusCommissionEmployee constructor
18
```

```
19 // set base salary
20 void BasePlusCommissionEmployee::setBaseSalary( double salary )
21 {
22    baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
23 } // end function setBaseSalary
24
25 // return base salary
26 double BasePlusCommissionEmployee::getBaseSalary() const
27 {
28    return baseSalary;
29 } // end function getBaseSalary
30
31 // calculate earnings
32 double BasePlusCommissionEmployee::earnings() const
33 {
34    // can access protected data of base class
35    return baseSalary + ( commissionRate * grossSales );
36 } // end function earnings
37
38 // print BasePlusCommissionEmployee object
39 void BasePlusCommissionEmployee::print() const
40 {
41    // can access protected data of base class
42    cout << "base-salaried commission employee: " << firstName << ' '
43       << lastName << "\nsocial security number: " << socialSecurityNumber
44       << "\ngross sales: " << grossSales
45       << "\ncommission rate: " << commissionRate
46       << "\nbase salary: " << baseSalary;
47 } // end function print
```

شکل ۱۵-۱۲ | فایل پیاده‌سازی کلاس BasePlusCommssionEmployee.

تست کلاس اصلاح شده BasePlusCommssionEmployee

در برنامه ۱۶-۱۲ از یک شی **BasePlusCommssionEmployee** برای انجام همان وظایف که برنامه ۹-۱۲ بر روی یک شی از نسخه اول کلاس **BasePlusCommssionEmployee** انجام می‌داد (شکل‌های ۷-۱۲ و ۸-۱۲) استفاده شده است. دقت کنید که خروجی هر دو برنامه یکسان هستند. ابتدا **BasePlusCommssionEmployee** را بدون استفاده از توارث ایجاد کرده و این نسخه جدید را با استفاده از ارث‌بری ایجاد کرده‌ایم. با این همه هر دو کلاس وظایف یکسانی را انجام می‌دهند. توجه کنید که کد کلاس **BasePlusCommssionEmployee** (یعنی فایلهای سرآیند و پیاده‌سازی)، که 74 خط می‌شود. بطور قابل ملاحظه کوتاهتر از کد نسخه غیر ارث‌بر این کلاس می‌باشد که از 154 خط تشکیل شده است، چرا که نسخه ارث‌بر بخشی از قابلیت‌ها و وظایف خود را از **CommssionEmployee** به ارث برده و نسخه غیر ارث‌بر چنین خاصیتی ندارد. همچنین، در اینجا فقط یک کپی از توابع کلاس **CommissionEmployee** اعلان و تعریف شده است. در اینحالت نگهداری کد منبع، اصلاح و خطایابی آن آسانتر می‌شود، چرا که کد منبع مرتبط با **CommisionEmployee** فقط در فایلهای شکل ۱۲-۱۲ و ۱۳-۱۲ قرار دارند.

```
1  // Fig. 12.16: fig12_16.cpp
2  // Testing class BasePlusCommissionEmployee.
3  #include <iostream>
4  using std::cout;
```

```
5  using std::endl;
6  using std::fixed;
7
8  #include <iomanip>
9  using std::setprecision;
10
11 // BasePlusCommissionEmployee class definition
12 #include "BasePlusCommissionEmployee.h"
13
14 int main()
15 {
16    // instantiate BasePlusCommissionEmployee object
17    BasePlusCommissionEmployee
18       employee( "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
19
20    // set floating-point output formatting
21    cout << fixed << setprecision( 2 );
22
23    // get commission employee data
24    cout << "Employee information obtained by get functions: \n"
25       << "\nFirst name is " << employee.getFirstName()
26       << "\nLast name is " << employee.getLastName()
27       << "\nSocial security number is "
28       << employee.getSocialSecurityNumber()
29       << "\nGross sales is " << employee.getGrossSales()
30       << "\nCommission rate is " << employee.getCommissionRate()
31       << "\nBase salary is " << employee.getBaseSalary() << endl;
32
33    employee.setBaseSalary( 1000 ); // set base salary
34
35    cout << "\nUpdated employee information output by print function: \n"
36       << endl;
37    employee.print(); // display the new employee information
38
39    // display the employee's earnings
40    cout << "\n\nEmployee's earnings: $" << employee.earnings() << endl;
41
42    return 0;
43 } // end main
```

```
Employee information obtained by get functions:

First name is Bob
Last name is Lewis
Social security number is 333-33-3333
Gross sales is 5000.00
Commission rate is 0.04
Base salary is 300.00

Update employee information output by print function:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 1000.00

Employee's earnings: $1200.00
```

شکل ۱۶-۱۲ داده کلاس مبنای protected می‌تواند از طریق کلاس مشتق شده در دسترس قرار گیرد.

نکاتی در ارتباط با استفاده از داده protected

در این مثال، اعضای داده کلاس مبنا را بصورت **protected** اعلان کردیم، از اینروست که کلاس‌های مشتق شده قادر به اصلاح داده‌ها بصورت مستقیم هستند. ارث‌بری اعضاء داده **protected**، کمی در افزایش کارایی موثر است، چرا که می‌توانیم مستقیماً به اعضا دسترسی پیدا کنیم بدون اینکه متحمل

فراخوانی‌های اضافی توابع عضو *get* یا *set* شده باشیم. با این وجود، در بسیاری از موارد، بهتر است از اعضای داده **private** استفاده کنیم تا به لحاظ مهندسی نرم‌افزار در مسیر درستی قرار داشته باشیم و وظیفه بهینه‌سازی کد را به کامپایلر واگذار کنیم. در اینحالت نگهداری، خطایابی و اصلاح برنامه آسانتر می‌شود.

استفاده از اعضای داده **protected** دو مشکل عمده دارد. اول اینکه، شی از کلاس مشتق شده نمی‌تواند از یک تابع برای تنظیم مقدار عضو داده **protected** کلاس مبنا استفاده کند. از اینرو، یک شی از کلاس مشتق شده می‌تواند یک مقدار نامعتبر به عضو داده **protected** تخصیص دهد، از اینرو شی در وضعیت غیرپایدار باقی می‌ماند. برای مثال عضو داده **grossSales** از کلاس **CommssionEmployee** که بصورت **protected** اعلان شده است، یک شی از کلاس مشتق شده (مثلاً **BasePlusCommssionEmployee**) می‌تواند یک مقدار منفی به **grossSales** تخصیص دهد. مشکل دوم در ارتباط با استفاده از اعضای داده **protected** این است که توابع عضو از کلاس مشتق شده بستگی به پیاده‌سازی کلاس مبنا دارند. در عمل، کلاس‌های مشتق شده بایستی فقط به سرویس‌های کلاس مبنا بستگی داشته باشند (یعنی توابع عضو غیر **private**) و نه به پیاده‌سازی کلاس مبنا. در صورتی که اعضا داده در کلاس مبنا بصورت **protected** باشند و اگر پیاده‌سازی کلاس مبنا دچار تغییر شود، نیاز به اصلاح تمام کلاس‌های مشتق شده از آن کلاس مبنا خواهیم داشت. برای مثال، اگر به برخی از دلایل نیاز باشد که اسامی اعضای داده **firstName** و **lastName** را به **first** و **last** تغییر دهیم، مجبور هستیم در تمام مکان‌های که یک کلاس مشتق شده بصورت مستقیم به این اعضای کلاس مبنا مراجعه می‌کند این تغییرات را اعمال کنیم.

در چنین حالتی، گفته می‌شود که نرم‌افزار شکننده یا بی‌دوام است، چرا که یک تغییر کوچک در کلاس مبنا می‌تواند پیاده‌سازی کلاس مشتق شده را در هم ریزد.

۵-٤-۱۲ ایجاد سلسله مراتب توارث CommissionEmplyee-BasePlusCommssionEmployee با استفاده از private

اکنون باز هم به سراغ سلسله مراتب قبلی می‌رویم، اما این بار از یک روش مناسب در مهندسی نرم‌افزار استفاده خواهیم کرد. اعضای داده کلاس **CommsionEmployee** را بصورت **private** اعلان می‌کنیم (شکل ۱۷-۱۲، خطوط خطوط 37-33) و توابع عضو آنرا بصورت **public** در نظر می‌گیریم تا بتوانیم این مقادیر را نگهداری کنیم. اگر تصمیم به تغییر اسامی داده عضو بگیریم، دیگر تعاریف توابع **print** و **earnings** دچار تغییر نخواهند شد و فقط تعاریف توابع عضو *set* و *get* که مستقیماً با اعضای داده کار می‌کنند نیاز به تغییر خواهند داشت.

توجه کنید که این تغییرات منحصراً در درون کلاس مبنا صورت می‌گیرد و نیاز به اعمال هیچ تغییری در کلاس مشتق شده نخواهد بود. کلاس مشتق شده **BasePlusCommssionEmployee** (شکل‌های ۱۲-۱۹ و ۱۲-۲۰) توابع عضو غیر **private** را از کلاس **CommissionEmployee** به ارث برده و می‌تواند به اعضای **private** کلاس مبنا از طریق آن توابع دسترسی پیدا کند.

```
1  // Fig. 12.17: CommissionEmployee.h
2  // CommissionEmployee class definition with good software engineering.
3  #ifndef COMMISSION_H
4  #define COMMISSION_H
5
6  #include <string> // C++ standard string class
7  using std::string;
8
9  class CommissionEmployee
10 {
11 public:
12    CommissionEmployee( const string &, const string &, const string &,
13       double = 0.0, double = 0.0 );
14
15    void setFirstName( const string & ); // set first name
16    string getFirstName() const; // return first name
17
18    void setLastName( const string & ); // set last name
19    string getLastName() const; // return last name
20
21    void setSocialSecurityNumber( const string & ); // set SSN
22    string getSocialSecurityNumber() const; // return SSN
23
24    void setGrossSales( double ); // set gross sales amount
25    double getGrossSales() const; // return gross sales amount
26
27    void setCommissionRate( double ); // set commission rate
28    double getCommissionRate() const; // return commission rate
29
30    double earnings() const; // calculate earnings
31    void print() const; // print CommissionEmployee object
32 private:
33    string firstName;
34    string lastName;
35    string socialSecurityNumber;
36    double grossSales; // gross weekly sales
37    double commissionRate; // commission percentage
38 }; // end class CommissionEmployee
39
40 #endif
```

شکل ۱۲-۱۷ | تعریف کلاس CommissionEmployee به روش مناسب مهندسی نرم‌افزار.

```
1  // Fig. 12.18: CommissionEmployee.cpp
2  // Class CommissionEmployee member-function definitions.
3  #include <iostream>
4  using std::cout;
5
6  #include "CommissionEmployee.h" // CommissionEmployee class definition
7
8  // constructor
9  CommissionEmployee::CommissionEmployee(
10    const string &first, const string &last, const string &ssn,
11    double sales, double rate )
12    : firstName( first ), lastName( last ), socialSecurityNumber( ssn )
13 {
14    setGrossSales( sales ); // validate and store gross sales
15    setCommissionRate( rate ); // validate and store commission rate
16 } // end CommissionEmployee constructor
```

```
17
18 // set first name
19 void CommissionEmployee::setFirstName( const string &first )
20 {
21    firstName = first; // should validate
22 } // end function setFirstName
23
24 // return first name
25 string CommissionEmployee::getFirstName() const
26 {
27    return firstName;
28 } // end function getFirstName
29
30 // set last name
31 void CommissionEmployee::setLastName( const string &last )
32 {
33    lastName = last; // should validate
34 } // end function setLastName
35
36 // return last name
37 string CommissionEmployee::getLastName() const
38 {
39    return lastName;
40 } // end function getLastName
41
42 // set social security number
43 void CommissionEmployee::setSocialSecurityNumber( const string &ssn )
44 {
45    socialSecurityNumber = ssn; // should validate
46 } // end function setSocialSecurityNumber
47
48 // return social security number
49 string CommissionEmployee::getSocialSecurityNumber() const
50 {
51    return socialSecurityNumber;
52 } // end function getSocialSecurityNumber
53
54 // set gross sales amount
55 void CommissionEmployee::setGrossSales( double sales )
56 {
57    grossSales = ( sales < 0.0 ) ? 0.0 : sales;
58 } // end function setGrossSales
59
60 // return gross sales amount
61 double CommissionEmployee::getGrossSales() const
62 {
63    return grossSales;
64 } // end function getGrossSales
65
66 // set commission rate
67 void CommissionEmployee::setCommissionRate( double rate )
68 {
69    commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
70 } // end function setCommissionRate
71
72 // return commission rate
73 double CommissionEmployee::getCommissionRate() const
74 {
75    return commissionRate;
76 } // end function getCommissionRate
77
78 // calculate earnings
79 double CommissionEmployee::earnings() const
80 {
81    return getCommissionRate() * getGrossSales();
82 } // end function earnings
83
84 // print CommissionEmployee object
85 void CommissionEmployee::print() const
86 {
```

```
87    cout << "commission employee: "
88       << getFirstName() << ' ' << getLastName()
89       << "\nsocial security number: " << getSocialSecurityNumber()
90       << "\ngross sales: " << getGrossSales()
91       << "\ncommission rate: " << getCommissionRate();
92 } // end function print
```

شکل ۱۸-۱۲ | فایل پیاده‌سازی کلاس از CommissionEmployee.

در پیاده‌سازی **CommissionEmployee** (شکل ۱۸-۱۲، خطوط 16-9) توجه کنید که از مقداردهی کنند اولیه عضو (خط 12) برای تنظیم مقادیر عضو **firstName**، **lastName** و **socialSecurityNumber** استفاده کرده‌ایم. نشان داده‌ایم که چگونه کلاس مشتق شده **BasePlusCommssionEmployee** (شکل ۱۹-۱۲ و ۲۰-۱۲) می‌تواند توابع عضو کلاس مبنا را که غیر **private** هستند را برای کار با این اعضای داده فراخوانی کند.

کلاس **BasePlusCommssionEmployee** (شکل‌های ۱۹-۱۲ و ۲۰-۱۲) چندین تغییر در پیاده‌سازی تابع عضو خود دارد (شکل ۲۰-۱۲) که آنرا را از نسخه قبلی کلاس متمایز می‌سازد (شکل‌های ۱۴-۱۲ و ۱۵-۱۲).

```
1  // Fig. 12.19: BasePlusCommissionEmployee.h
2  // BasePlusCommissionEmployee class derived from class
3  // CommissionEmployee.
4  #ifndef BASEPLUS_H
5  #define BASEPLUS_H
6
7  #include <string> // C++ standard string class
8  using std::string;
9
10 #include "CommissionEmployee.h" // CommissionEmployee class declaration
11
12 class BasePlusCommissionEmployee : public CommissionEmployee
13 {
14 public:
15    BasePlusCommissionEmployee( const string &, const string &,
16       const string &, double = 0.0, double = 0.0, double = 0.0 );
17
18    void setBaseSalary( double ); // set base salary
19    double getBaseSalary() const; // return base salary
20
21    double earnings() const; // calculate earnings
22    void print() const; // print BasePlusCommissionEmployee object
23 private:
24    double baseSalary; // base salary
25 }; // end class BasePlusCommissionEmployee
26
27 #endif
```

شکل ۱۹-۱۲ | فایل سرآیند کلاس BasePlusCommssionEmployee.

```
1  // Fig. 12.20: BasePlusCommissionEmployee.cpp
2  // Class BasePlusCommissionEmployee member-function definitions.
3  #include <iostream>
4  using std::cout;
5
6  // BasePlusCommissionEmployee class definition
7  #include "BasePlusCommissionEmployee.h"
8
9  // constructor
```

```
10  BasePlusCommissionEmployee::BasePlusCommissionEmployee(
11     const string &first, const string &last, const string &ssn,
12     double sales, double rate, double salary )
13     // explicitly call base-class constructor
14     : CommissionEmployee( first, last, ssn, sales, rate )
15  {
16     setBaseSalary( salary ); // validate and store base salary
17  } // end BasePlusCommissionEmployee constructor
18
19  // set base salary
20  void BasePlusCommissionEmployee::setBaseSalary( double salary )
21  {
22     baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
23  } // end function setBaseSalary
24
25  // return base salary
26  double BasePlusCommissionEmployee::getBaseSalary() const
27  {
28     return baseSalary;
29  } // end function getBaseSalary
30
31  // calculate earnings
32  double BasePlusCommissionEmployee::earnings() const
33  {
34     return getBaseSalary() + CommissionEmployee::earnings();
35  } // end function earnings
36
37  // print BasePlusCommissionEmployee object
38  void BasePlusCommissionEmployee::print() const
39  {
40     cout << "base-salaried ";
41
42     // invoke CommissionEmployee's print function
43     CommissionEmployee::print();
44
45     cout << "\nbase salary: " << getBaseSalary();
46  } // end function print
```

شکل ۲۰-۱۲ | کلاس BasePlusCommssionEmployee که از کلاس CommissionEmployee ارث‌بری دارد اما نمی‌تواند مستقیماً به داده private کلاس دسترسی پیدا کند.

توابع عضو **earnings** (شکل ۲۰-۲، خطوط 35-32) و **print** (خطوط 46-38) هر یک تابع عضو **getBaseSalary** را برای بدست آوردن مقدار حقوق پایه، بجای دسترسی مستقیم به **baseSalary** را احضار می‌کنند. این روش از تغییرات **earnings** و **print** که در صورت تغییر در پیاده‌سازی عضو داده **baseSalary** رخ می‌دهد، حفاظت می‌کند. برای مثال، اگر تصمیم به تغییر نام دادن عضو داده **baseSalary** یا تغییر نوع آن بگیریم، فقط توابع عضو **setBaseSalary** و **getBaseSalary** نیاز به تغییر خواهند داشت.

تابع **earnings** (شکل ۲۰-۱۲، خطوط 35-32) تعریف مجددی از تابع عضو **earnings** از کلاس **CommissionEmployee** (شکل ۱۸-۱۲، خطوط 82-79) برای محاسبه حقوق برای کارمندی است که حقوق و کمیسیونی از فروش دریافت می‌کند. نسخه **earnings** از کلاس **BasePlusCommssionEmployee** بخشی از حقوق کارمند را بر مبنای کمیسیون را صرفاً با فراخوانی تابع **earnings** کلاس مبنا با عبارت **commissionEmployee::earnings()** بدست می‌آورد. (شکل ۲۲-

۱۲، خط 34). سپس تابع **earnings** از کلاس **BasePlusCommssionEmployee** اقدام به افزودن حقوق پایه به این مقدار می‌کند، تا کل حقوق کارمند محاسبه شود. به گرامر بکار رفته در فراخوانی یک تابع عضو کلاس مبنا که مجدداً تعریف شده از یک کلاس مشتق شده است دقت کنید. قرار دادن نام کلاس مبنا و عملگر باینری تفکیک قلمرو (::) قبل از نام تابع عضو کلاس مبنا. با داشتن تابع **earnings** کلاس **BasePlusCommssionEmployee** که تابع **earnings** از کلاس **CommissionEmployee** را برای محاسبه بخشی از حقوق شی از **BasePlusCommssionEmployee** فراخوانی می‌کنند، از تکثیر کد اجتناب شده و مشکلات نگهداری کد کاهش می‌یابد.

همین حالت برای تابع **BasePlusCommssionEmployee** (شکل ۲۰-۱۲، خطوط 46-38) که تعریف مجددی از تابع عضو **print** از کلاس **CommissionEmployee** است، صادق می‌باشد (شکل ۱۸-۱۲، خطوط 92-85). این تابع اطلاعاتی در ارتباط با کارمندی که حقوق پایه به همراه کمیسیون را دریافت می‌کند، به نمایش در می‌آورد.

برنامه شکل ۲۱-۱۲ همان کارها را بر روی یک شی **BasePlusCommssionEmployee** را همانند شکل‌های ۹-۱۲ و ۱۶-۱۲ که بر روی شی‌های از کلاس **CommissionEmployee** و **BasePlusCommssionEmployee** انجام می‌دادند، انجام می‌دهد. با استفاده از توارث و فراخوانی توابع عضو که داده‌ها در آنها پنهان است، کلاسی خواهیم داشت که بخوبی ایجاد شده و از کارایی مناسبی نیز برخوردار است.

```
1   // Fig. 12.21: fig12_21.cpp
2   // Testing class BasePlusCommissionEmployee.
3   #include <iostream>
4   using std::cout;
5   using std::endl;
6   using std::fixed;
7
8   #include <iomanip>
9   using std::setprecision;
10
11  // BasePlusCommissionEmployee class definition
12  #include "BasePlusCommissionEmployee.h"
13
14  int main()
15  {
16     // instantiate BasePlusCommissionEmployee object
17     BasePlusCommissionEmployee
18        employee( "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
19
20     // set floating-point output formatting
21     cout << fixed << setprecision( 2 );
22
23     // get commission employee data
24     cout << "Employee information obtained by get functions: \n"
25        << "\nFirst name is " << employee.getFirstName()
26        << "\nLast name is " << employee.getLastName()
27        << "\nSocial security number is "
28        << employee.getSocialSecurityNumber()
```

```
29          << "\nGross sales is " << employee.getGrossSales()
30          << "\nCommission rate is " << employee.getCommissionRate()
31          << "\nBase salary is " << employee.getBaseSalary() << endl;
32
33       employee.setBaseSalary( 1000 ); // set base salary
34
35       cout << "\nUpdated employee information output by print function: \n"
36          << endl;
37       employee.print(); // display the new employee information
38
39       // display the employee's earnings
40       cout << "\n\nEmployee's earnings: $" << employee.earnings() << endl;
41
42       return 0;
43    } // end main
```

```
Employee information obtained by get functions:

First name is Bob
Last name is Lewis
Social security number is 333-33-3333
Gross sales is 5000.00
Commission rate is 0.04
Base salary is 300.00

Update employee information output by print function:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 1000.00

Employee's earnings: $1200.00
```

شکل ۱۲-۲۱ | داده private کلاس مبنا که برای یک کلاس مشتق شده از طریق تابع عضو public یا protected ارث رفته توسط کلاس مشتق شده در دسترس می باشد.

۱۲-۵ سازنده‌ها و پایان‌دهنده‌ها در کلاس‌های مشتق شده

همانطوری که در بخش‌های قبلی گفته شد، نمونه‌سازی یک شی کلاس مشتق شده با فراخوانی سازنده‌های کلاس مبنا صورت می‌گیرد و اینکار قبل از آنکه سازنده‌های مشتق شده قادر به انجام وظایف خود باشند اعمال می‌شود. فراخوانی سازنده کلاس مبنا می‌تواند بصورت صریح و غیرصریح انجام شود. بطور مشابه اگر کلاس مبنا از کلاس دیگری مشتق شده باشد، بایستی سازنده کلاس مبنا اقدام به فراخوانی سازنده کلاس بعدی در درخت سلسله مراتب نماید و اینکار تا پایان ادامه می‌یابد. آخرین سازنده فراخوانی شده در این زنجیره سازنده کلاس در بالای سلسله مراتب است که ابتدا اجرای بدنه آن خاتمه می‌یابد. هر سازنده کلاس مبنا اعضای داده کلاس مبنا را که توسط کلاس‌های مشتق شده به ارث برده شده‌اند، مقداردهی اولیه می‌کند. برای مثال، به سلسله مراتب **CommissionEmployee/BasePlusCommissionEmployee** در شکل‌های ۱۲-۱۷ الی ۱۲-۲۰ توجه نمائید. هنگامی که برنامه اقدام به ایجاد یک شی **BasePlusCommissionEmployee** می‌کند، یکی از سازنده‌های **CommissionEmployee** فراخوانی می‌شود.

از آنجایی که کلاس **CommissionEmployee** در بالای سلسله مراتب قرار دارد، سازنده آن اجرا شده، اعضای داده **private** آن که بخشی از شی **BasePlusCommssionEmployee** می‌باشند مقداردهی اولیه می‌شوند. زمانیکه اجرای سازنده **CommissionEmployee** کامل شده، کنترل را به سازنده **BasePlusCommssionEmployee** برگشت می‌دهد، که آن هم **baseSalary** را مقداردهی اولیه می‌نماید.

زمانیکه یک شی از کلاس مشتق شده نابود می‌شود، برنامه، نابود کننده آن شی را فراخوانی می‌کند. اینکار با فراخوانی زنجیره‌وار نابود کننده‌ها شروع می‌شود که در آن نابود کننده کلاس مشتق شده و نابود کننده‌های مستقیم و غیرمستقیم کلاس‌های مبنا و اعضای کلاس‌ها به ترتیب معکوس از اجرای سازنده‌ها، اجرا می‌شوند. زمانیکه نابود کننده یک شی کلاس مشتق شده فراخوانی می‌گردد، نابود کننده وظیفه خود را انجام می‌دهد، سپس نابود کننده‌ای را که در یک سطح بالاتر از سلسله مراتب قرار دارد، احضار می‌کند. این فرآیند تا فراخوانی نابود کننده قرار گرفته در بالاترین سطح سلسله مراتب ادامه می‌یابد. سپس شی از حافظه حذف می‌گردد.

سازنده‌ها، نابود کننده‌ها و عملگرهای سربارگذاری شده تخصیص کلاس مبنا توسط کلاس‌های مشتق شده، ارث‌بری نمی‌شوند. با این وجود، سازنده‌ها، نابود کننده‌ها و عملگرهای تخصیص سربارگذاری شده کلاس مشتق شده می‌توانند سازنده‌ها، نابود کننده‌ها و عملگرهای تخصیص سربارگذاری شده کلاس مبنا را فراخوانی کنند.

مثال بعدی نگاهی مجدد به سلسله مراتب کارمند کمیسیون بگیر است که توسط کلاس **CommissionEmployee** (شکل‌های۲۲-۱۲ و ۲۳-۱۲) و کلاس **BasePlusCommssionEmployee** (شکل‌های ۲۴-۱۲ و ۲۵-۱۲) تعریف شده و حاوی سازنده‌ها و نابود کننده‌های است که هر یک به هنگام فراخوانی پیغامی چاپ می‌کنند. همانطوری که در خروجی شکل ۲۶-۱۲ مشاهده می‌کنید، این پیغام‌ها ترتیب فراخوانی سازنده‌ها و نابود کننده‌ها را در سلسله مراتب توارث نشان می‌دهند.

```
// Fig. 12.22: CommissionEmployee.h
// CommissionEmployee class definition represents a commission employee.
#ifndef COMMISSION_H
#define COMMISSION_H

#include <string> // C++ standard string class
using std::string;

class CommissionEmployee
{
public:
   CommissionEmployee( const string &, const string &, const string &,
      double = 0.0, double = 0.0 );
   ~CommissionEmployee(); // destructor

```

```
16      void setFirstName( const string & ); // set first name
17      string getFirstName() const; // return first name
18
19      void setLastName( const string & ); // set last name
20      string getLastName() const; // return last name
21
22      void setSocialSecurityNumber( const string & ); // set SSN
23      string getSocialSecurityNumber() const; // return SSN
24
25      void setGrossSales( double ); // set gross sales amount
26      double getGrossSales() const; // return gross sales amount
27
28      void setCommissionRate( double ); // set commission rate
29      double getCommissionRate() const; // return commission rate
30
31      double earnings() const; // calculate earnings
32      void print() const; // print CommissionEmployee object
33   private:
34      string firstName;
35      string lastName;
36      string socialSecurityNumber;
37      double grossSales; // gross weekly sales
38      double commissionRate; // commission percentage
39   }; // end class CommissionEmployee
40
41   #endif
```

شکل ۲۲-۱۲ | فایل سرآیند کلاس CommissionEmployee.

در این مثال، سازنده **CommissionEmployee** را اصلاح کرده (خطوط 21-10 از شکل ۲۳-۱۲) و یک نابود کننده **CommissionEmployee** (خطوط 29-24) به آن افزوده‌ایم، که هر کدام به هنگام فراخوانی یک پیغام مناسب در خروجی قرار می‌دهند.همچنین سازنده **BasePlusCommssionEmployee** را اصلاح کرده (خطوط 22-11 از شکل ۲۵-۱۲) و یک نابود کننده به آن افزوده‌ایم (خطوط 30-25) که هر کدام به هنگام فراخوانی یک پیغام مناسب در خروجی قرار می‌دهند.

```
1    // Fig. 12.23: CommissionEmployee.cpp
2    // Class CommissionEmployee member-function definitions.
3    #include <iostream>
4    using std::cout;
5    using std::endl;
6
7    #include "CommissionEmployee.h" // CommissionEmployee class definition
8
9    // constructor
10   CommissionEmployee::CommissionEmployee(
11      const string &first, const string &last, const string &ssn,
12      double sales, double rate )
13      : firstName( first ), lastName( last ), socialSecurityNumber( ssn )
14   {
15      setGrossSales( sales ); // validate and store gross sales
16      setCommissionRate( rate ); // validate and store commission rate
17
18      cout << "CommissionEmployee constructor: " << endl;
19      print();
20      cout << "\n\n";
21   } // end CommissionEmployee constructor
22
23   // destructor
24   CommissionEmployee::~CommissionEmployee()
25   {
26      cout << "CommissionEmployee destructor: " << endl;
27      print();
28      cout << "\n\n";
```

```
29  } // end CommissionEmployee destructor
30
31  // set first name
32  void CommissionEmployee::setFirstName( const string &first )
33  {
34     firstName = first; // should validate
35  } // end function setFirstName
36
37  // return first name
38  string CommissionEmployee::getFirstName() const
39  {
40     return firstName;
41  } // end function getFirstName
42
43  // set last name
44  void CommissionEmployee::setLastName( const string &last )
45  {
46     lastName = last; // should validate
47  } // end function setLastName
48
49  // return last name
50  string CommissionEmployee::getLastName() const
51  {
52     return lastName;
53  } // end function getLastName
54
55  // set social security number
56  void CommissionEmployee::setSocialSecurityNumber( const string &ssn )
57  {
58     socialSecurityNumber = ssn; // should validate
59  } // end function setSocialSecurityNumber
60
61  // return social security number
62  string CommissionEmployee::getSocialSecurityNumber() const
63  {
64     return socialSecurityNumber;
65  } // end function getSocialSecurityNumber
66
67  // set gross sales amount
68  void CommissionEmployee::setGrossSales( double sales )
69  {
70     grossSales = ( sales < 0.0 ) ? 0.0 : sales;
71  } // end function setGrossSales
72
73  // return gross sales amount
74  double CommissionEmployee::getGrossSales() const
75  {
76     return grossSales;
77  } // end function getGrossSales
78
79  // set commission rate
80  void CommissionEmployee::setCommissionRate( double rate )
81  {
82     commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
83  } // end function setCommissionRate
84
85  // return commission rate
86  double CommissionEmployee::getCommissionRate() const
87  {
88     return commissionRate;
89  } // end function getCommissionRate
90
91  // calculate earnings
92  double CommissionEmployee::earnings() const
93  {
94     return getCommissionRate() * getGrossSales();
95  } // end function earnings
96
97  // print CommissionEmployee object
98  void CommissionEmployee::print() const
```

```
99  {
100    cout << "commission employee: "
101       << getFirstName() << ' ' << getLastName()
102       << "\nsocial security number: " << getSocialSecurityNumber()
103       << "\ngross sales: " << getGrossSales()
104       << "\ncommission rate: " << getCommissionRate();
105 } // end function print
```

شکل ۲۳-۱۲ | سازنده CommissionEmployee که متنی در خروجی قرار می‌دهد.

```
1   // Fig. 12.24: BasePlusCommissionEmployee.h
2   // BasePlusCommissionEmployee class derived from class
3   // CommissionEmployee.
4   #ifndef BASEPLUS_H
5   #define BASEPLUS_H
6
7   #include <string> // C++ standard string class
8   using std::string;
9
10  #include "CommissionEmployee.h" // CommissionEmployee class declaration
11
12  class BasePlusCommissionEmployee : public CommissionEmployee
13  {
14  public:
15     BasePlusCommissionEmployee( const string &, const string &,
16        const string &, double = 0.0, double = 0.0, double = 0.0 );
17     ~BasePlusCommissionEmployee(); // destructor
18
19     void setBaseSalary( double ); // set base salary
20     double getBaseSalary() const; // return base salary
21
22     double earnings() const; // calculate earnings
23     void print() const; // print BasePlusCommissionEmployee object
24  private:
25     double baseSalary; // base salary
26  }; // end class BasePlusCommissionEmployee
27
28  #endif
```

شکل ۲۴-۱۲ | فایل سرآیند کلاس BasePlusCommssionEmployee.

```
1   // Fig. 12.25: BasePlusCommissionEmployee.cpp
2   // Class BasePlusCommissionEmployee member-function definitions.
3   #include <iostream>
4   using std::cout;
5   using std::endl;
6
7   // BasePlusCommissionEmployee class definition
8   #include "BasePlusCommissionEmployee.h"
9
10  // constructor
11  BasePlusCommissionEmployee::BasePlusCommissionEmployee(
12     const string &first, const string &last, const string &ssn,
13     double sales, double rate, double salary )
14     // explicitly call base-class constructor
15     : CommissionEmployee( first, last, ssn, sales, rate )
16  {
17     setBaseSalary( salary ); // validate and store base salary
18
19     cout << "BasePlusCommissionEmployee constructor: " << endl;
20     print();
21     cout << "\n\n";
22  } // end BasePlusCommissionEmployee constructor
23
24  // destructor
25  BasePlusCommissionEmployee::~BasePlusCommissionEmployee()
26  {
27     cout << "BasePlusCommissionEmployee destructor: " << endl;
28     print();
```

```
29       cout << "\n\n";
30    } // end BasePlusCommissionEmployee destructor
31
32    // set base salary
33    void BasePlusCommissionEmployee::setBaseSalary( double salary )
34    {
35       baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
36    } // end function setBaseSalary
37
38    // return base salary
39    double BasePlusCommissionEmployee::getBaseSalary() const
40    {
41       return baseSalary;
42    } // end function getBaseSalary
43
44    // calculate earnings
45    double BasePlusCommissionEmployee::earnings() const
46    {
47       return getBaseSalary() + CommissionEmployee::earnings();
48    } // end function earnings
49
50    // print BasePlusCommissionEmployee object
51    void BasePlusCommissionEmployee::print() const
52    {
53       cout << "base-salaried ";
54
55       // invoke CommissionEmployee's print function
56       CommissionEmployee::print();
57
58       cout << "\nbase salary: " << getBaseSalary();
59    } // end function print
```

شکل ۲۵-۱۲ | سازنده BasePlusCommssionEmployee که متنی در خروجی قرار می‌دهد.

برنامه شکل ۲۶-۱۲ به توصیف ترتیب فراخوانی سازنده‌ها و نابود کننده‌ها برای شی‌هایی می‌پردازد که بخشی از یک سلسله مراتب هستند. تابع **main** (خطوط 34-15) با نمونه‌سازی شی **employee1** از **CommissionEmployee** (خطوط 22-21) در یک بلوک مجزا در درون **main** شروع می‌شود (خطوط 23-20).شی بلافاصله به خارج از قلمرو خود می‌رود،از اینرو سازنده و نابود کننده **CommissionEmployee** فراخوانی می‌شوند. سپس، خطوط 27-26 شی **Employee2** از **BasePlusCommssion** را ایجاد می‌کنند. با اینکار سازنده **CommissionEmployee** برای نمایش خروجی با مقادیر ارسالی از سازنده **BasePlusCommssionEmployee** فراخوانی شده، سپس خروجی تعیین شده در سازنده **BasePlusCommssionEmployee** به کار می‌افتد. سپس خطوط 31-30 مبادرت به ایجاد شی **employee3** از **BasePlusCommssionEmployee** می‌کنند. مجدداً هر دو سازنده **CommissionEmployee** و **BasePlusCommssionEmployee** فراخوانی می‌شوند. توجه کنید که در هر دو مورد، بدنه سازنده **CommissionEmployee** قبل از بدنه سازنده **BasePlusCommssionEmployee** اجرا می‌شود. زمانیکه به انتهای **main** می‌رسیم، نابود کننده‌ها برای شی‌های **employee2** و **employee3** فراخوانی می‌شوند.

اما بدلیل اینکه فراخوانی نابود کننده‌ها به ترتیب عکس از سازنده‌های متناظر با آنها صورت می‌گیرد، نابود کننده **BasePlusCommssionEmployee** و نابود کننده **CommssionEmployee** برای شی **employee3** فراخوانی شده و سپس نابود کننده‌های **BasePlusCommssionEmployee** و **CommissionEmplyee** برای شی **employee2** فراخوانی می‌شوند.

```
1    // Fig. 12.26: fig12_26.cpp
2    // Display order in which base-class and derived-class constructors
3    // and destructors are called.
4    #include <iostream>
5    using std::cout;
6    using std::endl;
7    using std::fixed;
8
9    #include <iomanip>
10   using std::setprecision;
11
12   // BasePlusCommissionEmployee class definition
13   #include "BasePlusCommissionEmployee.h"
14
15   int main()
16   {
17      // set floating-point output formatting
18      cout << fixed << setprecision( 2 );
19
20      { // begin new scope
21         CommissionEmployee employee1(
22            "Bob", "Lewis", "333-33-3333", 5000, .04 );
23      } // end scope
24
25      cout << endl;
26      BasePlusCommissionEmployee
27         employee2( "Lisa", "Jones", "555-55-5555", 2000, .06, 800 );
28
29      cout << endl;
30      BasePlusCommissionEmployee
31         employee3( "Mark", "Sands", "888-88-8888", 8000, .15, 2000 );
32      cout << endl;
33      return 0;
34   } // end main
```

```
CommissionEmployee constructor:
commission employee:Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04

CommissionEmployee destructor:
commission employee:Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04

CommissionEmployee constructor:
base-salaried commission employee: Lisa Jones
social security number: 555-55-5555
gross sales: 2000.00
commission rate: 0.06

BasePluCommissionEmployee constructor:
base-salaried commission employee: Lisa Jones
social security number: 555-55-5555
gross sales: 2000.00
commission rate: 0.06
base salary: 800.00
```

```
CommissionEmployee constructor:
commission employee: Mark Sands
social security number: 888-88-8888
gross sales: 8000.00
commission rate: 0.15

BasePluCommissionEmployee constructor:
base-salaried commission employee: Mark Sands
social security number: 888-88-8888
gross sales: 8000.00
commission rate: 0.15
base salary: 2000.00

BasePluCommissionEmployee destructor:
base-salaried commission employee: Mark Sands
social security number: 888-88-8888
gross sales: 8000.00
commission rate: 0.15
base salary: 2000.00

CommissionEmployee destructor:
commission employee: Mark Sands
social security number: 888-88-8888
gross sales: 8000.00
commission rate: 0.15

BasePluCommissionEmployee destructor:
base-salaried commission employee: Lisa Jones
social security number: 555-55-5555
gross sales: 2000.00
commission rate: 0.06
base salary: 800.00

CommissionEmployee destructor:
base-salaried commission employee: Lisa Jones
social security number: 555-55-5555
gross sales: 2000.00
commission rate: 0.06
```

شکل ۲۶-۱۲ | ترتیب فراخوانی سازنده و نابود کننده.

۶-۱۲ توارث **public**، **proteted** و **private**

زمانیکه کلاسی از یک کلاس مبنا، مشتق می‌شود، کلاس مبنا می‌تواند از طریق ارثبری **public**، **protected** و **private** به ارث برود. استفاده از روش ارث‌بری **protected** و **private** بندرت اتفاق می‌افتد و در استفاده از آنها باید دقت کرد. در این کتاب ما از روش توارث **public** استفاده می‌کنیم. جدول شکل ۲۷-۱۲ هر یک از انواع توارث و میزان ارث‌بری از طریق یک کلاس مشتق شده را بطور خلاصه عرضه کرده است. ستون اول حاوی تصریح کننده‌های دسترسی کلاس مبنا است.

به هنگام مشتق کردن یک کلاس از یک کلاس مبنای **public**، اعضای **public** کلاس مبنا، تبدیل به اعضای **public** کلاس مشتق شده گردیده و اعضای **protected** از کلاس مبنا، تبدیل به اعضای **protected** کلاس مشتق شده می‌شوند. اعضای **private** کلاس مبنا هرگز بطور مستقیم از طریق یک کلاس مشتق شده در دسترس نمی‌باشند، اما می‌توان آنها را از طریق فراخوانی اعضای **public** و **protected** کلاس مبنا دسترسی پیدا کرد.

به هنگام مشتق کردن یک کلاس از یک کلاس مبنای **protected**، اعضای **public** و **protected** کلاس مبنا، تبدیل به اعضای **protected** کلاس مشتق شده می‌شوند. به هنگام مشتق کردن یک کلاس از یک کلاس مبنای **private**، اعضای **public** و **protected** از کلاس مبنا، تبدیل به اعضای **private** کلاس مشتق می‌شوند (یعنی توابع تبدیل به توابع یوتیلیتی می‌شوند) توارث **private** و **protected** یک رابط is-a (است–یک) نیست.

تصریح کننده دسترسی عضو کلاس مبنا	نوع توارث: توارث **public**	نوع توارث: توارث **protected**	نوع توارث: توارث **private**
public	**public** در کلاس مشتق شده. می‌تواند مستقیماً توسط توابع عضو، توابع **friend** و توابع غیر عضو در دسترس قرار گیرد	**protected** در کلاس مشتق شده. می‌تواند مستقیماً توسط توابع عضو و توابع **friend** در دسترس قرار گیرد.	**private** در کلاس مشتق شده می‌تواند مستقیماً توسط توابع عضو و توابع **friend** در دسترس قرار گیرد.
Protected	**protected** در کلاس مشتق شده. می‌تواند مستقیماً توسط عضو و توابع **friend** در دسترس قرار گیرد.	**protected** در کلاس مشتق شده. می‌تواند مستقیماً توسط توابع عضو و توابع **friend** در دسترس قرار گیرد.	**private** در کلاس مشتق شدند می‌تواند مستقیماً توسط توابع عضو و توابع **firend** در دسترس قرار گیرد.
private	پنهان در کلاس مشتق شده. می‌تواند توسط توابع عضو و توابع **friend** از طریق توابع عضو **public** یا **protected** کلاس مبنا در دسترس قرار گیرد.	پنهان در کلاس مشتق شده. می تواند توسط توابع عضو و توابع **friend** از طریق توابع عضو **public** یا **protected** کلاس مبنا در دسترس قرار گیرد.	پنهان در کلاس مشتق شده. می‌تواند توسط توابع عضو و توابع **friend** از طریق توابع عضو **public** یا **protected** کلاس مبنا در دسترس قرار گیرد.

شکل ۲۷-۱۲ | خلاصه‌ای از دسترسی به اعضای کلاس مبنا در یک کلاس مشتق شده.

۱۲-۷ مهندسی نرم‌افزار بکمک توارث

در این بخش به نقش بکارگیری توارث در بهینه‌سازی نرم‌افزار موجود می‌پردازیم. هنگامی که از توارث برای ایجاد یک کلاس از روی یک کلاس موجود استفاده می کنیم، کلاس جدید تعدادی از اعضای داده، توابع عضو کلاس موجود را به ارث می‌برد، همانطوری که در جدول شکل ۲۷-۱۲ توضیح داده شده

است. زمانیکه کلاس ایجاد شد، می‌توانیم با توجه به نیازهای خود اقدام به افزودن اعضا برای بهینه‌سازی کلاس جدید کنیم.

گاهی اوقات، درک مشکلاتی که طراحان مشغول بکار در پروژه‌های بزرگ و صنایع با آنها مواجه هستند، برای دانشجویان سخت است. اشخاصی که تجربه کار در چنین پروژه‌های را دارند معتقد هستند که بکارگیری مجدد نرم‌افزار می‌تواند نقش بسیار موثری در فرآیند توسعه نرم‌افزار داشته باشد. برنامه‌نویسی شی‌گرا امر بکارگیری مجدد نرم‌افزار را تسهیل بخشیده و از اینرو زمان توسعه کاهش می‌یابد.

فصل
سیزدهم

برنامه‌نویسی شیی‌گرا: چندریختی

اهداف

- چند ریختی چیست و چگونه می‌تواند در برنامه‌نویسی موثر بکار گرفته شود.
- اعلان و استفاده از توابع virtual در موثرتر کردن چندریختی.
- وجه تمایز مابین کلاس های انتزاعی و مقید.
- اعلان توابع virtual محض برای ایجاد کلاس‌های انتزاعی.
- نحوه استفاده از اطلاعات نوع زمان اجرا (RTTI) به همراه تبدیل نوع dynamic_cast، typeid و type_info.
- نحوه پیاده‌سازی توابع virtual توسط C++.
- نحوه استفاده از نابود کننده‌های virtual برای حصول اطمینان از اجرای تمام نابود کننده‌های مورد نیاز بر روی یک شی.

رئوس مطالب

۱-۱۳ مقدمه

در فصل‌های ۱۲-۹ در ارتباط با مباحث کلیدی برنامه‌نویسی شی‌گرا و تکنولوژی‌های آن شامل کلاس‌ها، شی‌ها، کپسوله‌سازی، سربارگذاری عملگر و توارث صحبت کردیم. حال به آموزش OOP با توضیح و تفسیر مفهوم *چندریختی* (*polymorphism*) در سلسله مراتب توارث ادامه می‌دهیم. چندریختی امکان می‌دهد تا برنامه‌ها بجای اینکه «*برنامه خاصی*» باشند، حالت یک «*برنامه کلی*» داشته باشند.

در عمل، چند ریختی امکان می‌دهد تا برنامه‌هایی بنویسیم که مبادرت به پردازش شی‌ها از کلاس‌هایی کنند که بخشی از همان سلسله مراتب کلاس هستند، همچنانکه همگی آنها شی‌های از سلسله مراتب کلاس مبنا می‌باشند. همانطوری که بزودی خواهید دید، چند ریختی با *هندل‌های* (دستگیره‌های) اشاره‌گر کلاس مبنا و مراجعه‌های کلاس مبنا کاری ندارد و بر پایه نام هندل‌ها عمل می‌کند.

به مثالی در ارتباط با چند ریختی توجه کنید. فرض کنید می‌خواهیم برنامه‌ای بنویسیم که حرکت چند نوع حیوان را شبیه‌سازی کند. کلاس‌های Fish (ماهی)، Frog (قورباغه) و Bird (پرنده) نشان‌دهنده سه نوع حیوان تحت بررسی هستند. تصور کنید که هر یک از این کلاس‌ها از کلاس مبنای Animal ارث‌بری دارند، که حاوی یک تابع move بوده و موقعیت جاری حیوان را نگهداری می‌کند. هر کلاس مشتق شده تابع move را پیاده‌سازی می‌کند. برنامه مبادرت به نگهداری یک بردار (vector) از اشاره‌گرها به شی‌های از انواع کلاس‌های مشتق شده Animal می‌کند. برای شبیه‌سازی حرکت حیوانات، برنامه به هر شی در هر ثانیه یک پیغام بنام move ارسال می‌کند. با این وجود، هر نوع خاص از حیوان به این پیغام move (حرکت) به روش خود پاسخ می‌دهد، برای مثال ماهی قادر به شنا به میزان دو فوت، قورباغه قادر به پرش به میزان سه فوت و پرنده قادر به پرواز به میزان ده فوت است. برنامه بطور جامع یک پیغام (همان move) را به هر شی ارسال می‌کند، اما هر شی از نحوه اصلاح موقعیت خود براساس نوع حرکتی خود مطلع است و بر مبنای آن حرکت می‌کند. بر پایه اینکه هر شی از نحوه «انجام فعل صحیح» مطلع است، واکنش به فراخوانی تابع یکسان، مفهوم کلیدی چندریختی یا polymorphism است. پیغام یکسان (در این مورد move) که به انواع شی‌ها ارسال می‌شود، نتایج مختلفی بدنبال دارد و از اینرو نشاندهنده مفهوم چندریختی است.

به کمک چندریختی، می‌توانیم سیستم‌های را طراحی و پیاده‌سازی کنیم که گسترش و بسط‌پذیری آنها آسانتر است. کلاس‌های جدید می‌توانند با کمی تغییر یا اصلاح در بخش‌های عمومی برنامه، به آن افزوده شوند، مادامیکه کلاس‌های جدید بخشی از سلسله مراتب توارثی باشند که برنامه بطور جامع آنرا پردازش می‌کند. تنها بخش‌های از برنامه که باید برای تطبیق یافتن با کلاس‌های جدید تغییر داده شوند آنهایی هستند که نیاز دارند تا از وجود کلاس‌های جدید افزوده شده به سلسه مراتب مستقیماً مطلع گردند. برای مثال، اگر کلاس Tortoise (لاک‌پشت) را که از کلاس Animal ارث‌بری دارد را ایجاد کنیم (که می‌تواند به پیغام move به میزان یک اینچ حرکت یا خزیدن واکنش نشان دهد)، فقط نیاز است تا کلاس Tortoise و آن بخشی که یک نمونه از شی Tortoise را شبیه‌سازی می‌کند را بنویسیم.

با مطرح کردن مثال‌های سعی می‌کنیم تا درک مناسبی از مفهوم توابع **virtual** (مجازی) و *مقیدسازی دینامیکی* بوجود آوریم. که زیر ساخت‌های از تکنولوژی چند ریختی هستند. سپس به مطرح کردن یک مبحث آموزشی می‌پردازیم که در آن سلسله مراتب **Employee** از فصل دورازده بازبینی شده است. در مبحث آموزشی، یک «*واسط*» مشترک برای تمام کلاس‌های موجود در سلسله مراتب تعریف می‌کنیم. این واسط با قابلیت‌های مشترک در میان کارمندان، بعنوان کلاس مبنای انتزاعی **Employee** نامیده می‌شود، که از کلاس‌های **SalariedEmployee**، **HourlyEmployee** و **CommissionEmployee**

مستقیماً ارث‌بری دارد و کلاس **BasePlusCommissionEmployee** بصورت غیرمستقیم. بزودی شاهد این مطلب خواهید بود که چگونه کلاسی "انتزاعی" می‌شود و کلاسی "غیرانتزاعی".

در این سلسله مراتب، هر کارمندی دارای یک تابع **earnings** (حقوق) برای محاسبه حقوق هفتگی است. این توابع حقوق براساس نوع کارمند عمل می‌کنند، برای نمونه، کارمند **SalariedEmployee** یک حقوق هفتگی ثابت صرفنظر از ساعت کاری دریافت می‌کند، در حالیکه به کارمندی از نوع HourlyEmployee براساس ساعات کاری و اضافه کاری حقوق پرداخت می‌شود. نحوه پردازش هر کارمند را در حالت کلی نشان خواهیم داد که در آن از اشاره‌گرهای کلاس مبنا برای فراخوانی تابع **earnings** از میان چندین شی از کلاس مشتق شده استفاده می‌شود. در این روش، برنامه‌نویس نیاز به توجه به نوع فراخوانی تابع دارد، که می‌تواند برای اجرای چندین تابع مختلف براساس شی‌های اشاره شده توسط اشاره‌گرهای کلاس مبنا، بکار گرفته شود.

ویژگی کلیدی این فصل، بحث در ارتباط با چند ریختی، توابع **virtual** و مقیدسازی دینامیکی است، که در آن از یک دیاگرام برای توضیح اینکه چگونه چندریختی می‌تواند در C++ پیاده‌سازی شود، استفاده شده است.

از سلسله مراتب **Employee** برای شرح قابلیت‌های بنام *اطلاعات نوع زمان اجرا* (RTTI) و تبدیل دینامیکی استفاده مجدد خواهیم کرد که به برنامه امکان تعیین نوع یک شی در زمان اجرا را فراهم آورده و شی براساس آن عمل می‌کند.

۱۳-۲ چند مثال از چندریختی

در این بخش، در ارتباط با مثال‌های از چند ریختی بحث می‌کنیم. در چند ریختی، یک تابع می‌تواند اعمال متفاوتی را با توجه به نوع شی که تابع احضار می‌شود انجام دهد. اگر کلاس **Rectangle** (مستطیل) از کلاس **Quadrilateral** (چهارضلعی) است، پس یک شی مستطیل، نسخه بسیار خاصی از یک شی چهارضلعی است. بنابر این، هر عملیاتی (همانند محاسبه محیط یا مساحت) که می‌تواند بر روی یک شی از کلاس چهارضلعی بکار گرفته شود نیز می‌تواند بر روی شی از کلاس مستطیل پیاده گردد. البته چنین عملیاتی را می‌توان بر روی انواع چهارضلعی، همانند مربع‌ها، متوازی‌الاضلاع‌ها و ذوزنقه‌ها انجام داد. چند ریختی زمانی اتفاق می‌افتد که برنامه مبادرت به فراخوانی یک تابع **virtual** (مجازی) از طریق اشاره‌گر یا مراجعه کلاس مبنا (یعنی چهارضلعی) کند. C++ بصورت دینامیکی یا پویا (یعنی در زمان اجرا) تابع مناسب را برای کلاس انتخاب می‌کند (با توجه به شی که نمونه‌سازی شده است). در بخش ۳-۱۳ مثالی در این ارتباط به همراه کد آن آورده شده است.

بعنوان یک مثال دیگر، فرض کنید که یک بازی ویدئوی طراحی کرده‌ایم که شی‌های از نوع مختلف را نمونه‌سازی می‌کند، که در بر گیرنده شی‌های از کلاس‌های **Martian** (مریخی)، **Venutian** (ونوسی)، **Plutonian** (پلوتونی)، **SpaceShip** (سفینه فضایی) و **LaseBean** (پرتو لیزر) است. فرض کنید که هر کدامیک از این کلاس‌ها از کلاس مبنای مشترکی بنام **SpaceObject** ارث‌بری دارند، که حاوی تابع عضو **draw** است. هر کلاس مشتق شده، این تابع را به روش مقتضی برای آن کلاس پیاده‌سازی می‌کند. برنامه مدیریت صحنه مبادرت به نگهداری یک حامل (یک بردار یا **vector**) می‌کند که وظیفه آن حفظ اشاره‌گرهای **SpaceObject** به شی‌های از کلاس‌های مختلف است. برای نوسازی صحنه، مدیر صحنه در زمان‌های منظم به هر شی، پیغام یکسان **draw** را ارسال می‌کند. هر نوع از شی به یک طریق منحصر بفرد به این پیغام واکنش نشان می‌دهد. برای مثال، یک شی **Martian** می‌تواند خود را به رنگ قرمز با تعداد مشخصی آنتن ترسیم کند. یک شی **SpaceShip** می‌تواند خود را بصورت یک بشقاب پرنده نقره‌ای رنگ ترسیم نماید. یک شی **LaserBeam** می‌تواند خود را بصورت یک پرتو قرمز رنگ در امتداد صحنه ترسیم کند. مجدداً همان پیغام (در این مورد، **draw**) به انواع مختلفی از شی‌ها ارسال می‌شود و نتیجه آن بشکل‌های مختلف در می‌آید.

یک مدیر صحنه چند ریختی کار افزودن کلاس‌های جدید به یک سیستم را با حداقل تغییرات در کد فراهم می‌آورد. فرض کنید که می‌خواهیم شی‌های از کلاس **Mercurian** (عطارد) به بازی ویدئوی اضافه کنیم. برای انجام اینکار، بایستی یک کلاس **Mercurian** ایجاد کنیم که از **SpaceObject** ارث‌بری داشته باشد، اما تعریف متعلق بخود را از تابع عضو **draw** داشته باشد. سپس، زمانیکه اشاره‌گرهای به شی‌های از کلاس **Mercurian** در حامل ظاهر شوند، دیگر برنامه‌نویس نیازی به اصلاح کد مدیر صحنه نخواهد داشت. مدیر صحنه تابع عضو **draw** را برای هر شی در حامل فراخوانی می‌کند، صرفنظر از نوع شی، بنابر این شی‌های جدید **Mercurian** براحتی کار خود را انجام می‌دهند. از اینرو برنامه‌نویسان می‌توانند بدون هیچ تغییری در سیستم (بجز ایجاد و وارد کردن خود کلاس‌ها)، از چند ریختی برای تطبیق دادن کلاس‌های دیگر حتی آنهایی که در زمان ایجاد سیستم تصوری از آنها وجود نداشت، استفاده کنند.

۱۳-۳ رابطه مابین شی‌ها در سلسله مراتب توارث

در بخش ۴-۱۲ یک سلسله مراتب کلاس کارمند ایجاد کردیم که در آن کلاس **BasePlusCommissionEmployee** از کلاس **CommissinEmployee** ارث‌بری داشت. در فصل ۱۲ مثال‌های مطرح گردید که در آنها شی‌های **CommissinEmployee** و **BasePlusCommissionEmployee** با استفاده از اسامی شی‌ها مبادرت به احضار توابع عضو خود

می‌کردند. در این بخش به بررسی دقیق‌تر رابطه موجود در میان کلاس‌ها در سلسله مراتب می‌پردازیم. در چند بخش بعدی به معرفی دنباله‌ای از مثال‌ها خواهیم پرداخت که به بررسی عملکرد اشاره‌گرهای کلاس مبنا و کلاس مشتق شده می‌پردازند و همچنین نشان می‌دهند که چگونه می‌توان از این اشاره‌گرها در احضار توابع عضو استفاده کرد. در انتهای این بخش، به معرفی نحوه بدست گرفتن رفتار چند ریختی از اشاره‌گرهای کلاس مبنا که به شی‌های از کلاس مشتق شده اشاره دارند، خواهیم پرداخت.

۱-۳-۱۳ احضار توابع کلاس مبنا از طریق شی‌های کلاس مشتق شده

در مثال شکل‌های ۱-۱۳ الی ۵-۱۳ به بررسی سه روش در هدایت اشاره‌گرهای کلاس مبنا و اشاره‌گرهای کلاس مشتق شده بطرف شی‌های کلاس مبنا و شی‌های کلاس مشتق شده می‌پردازیم. دو روش اول بسیار سر راست هستند، یک اشاره‌گر کلاس مبنا را به طرف یک شی از کلاس مبنا هدایت کرده (و توابع کلاس مبنا احضار می‌شوند) و اشاره‌گر یک کلاس مشتق شده را به طرف یک شی کلاس مشتق شده (و توابع کلاس مشتق شده احضار می‌شوند) هدایت می‌کنیم. سپس، به بررسی رابطه موجود مابین کلاس‌های مشتق شده و کلاس‌های مبنا (یعنی رابطه is-a در سلسله مراتب) با هدایت یک اشاره‌گر کلاس مبنا بطرف یک شی کلاس مشتق شده می‌پردازیم (و نشان می‌دهیم که براستی قابلیت‌های کلاس مبنا در شی از کلاس مشتق شده وجود دارد).

از کلاس **CommissionEmployee** (شکل‌های ۱-۱۳ و ۲-۱۳) که در فصل ۱۲ توضیح دادیم استفاده می‌کنیم تا کارمندانی را معرفی کنیم که براساس درصدی از فروش به آنها حقوق پرداخت می‌شود. از کلاس **BasePlusCommissionEmployee** (شکل‌های ۳-۱۳ و ۴-۱۳) که در فصل ۱۲ توضیح داده‌ایم، به منظور معرفی کارمندانی که یک حقوق پایه به اضافه درصدی از فروش خود دریافت می‌کنند، استفاده می‌کنیم. هر شی **BasePlusCommissionEmployee** یک **CommissionEmployee** است (رابطه is-a) که دارای حقوق پایه نیز می‌باشد. کلاس **BasePlusCommissionEmployee** دارای تابع عضو **earnings** است (خطوط 35-32 از شکل ۴-۱۳) که تعریف مجددی از تابع عضو **earnings** از کلاس **CommissionEmployee** می‌باشد (خطوط 82-79 از شکل ۲-۱۳) که به آن حقوق پایه نیز افزوده گردیده است. تابع عضو **print** کلاس **BasePlusCommissionEmployee** (خطوط 46-38 از شکل ۴-۱۳) تعریف مجددی از تابع عضو **print** کلاس **CommissionEmployee** است (خطوط 92-85 از شکل ۲-۱۳) تا همان اطلاعات را به همراه حقوق پایه کارمند به نمایش در آورد.

```
1  // Fig. 13.1: CommissionEmployee.h
2  // CommissionEmployee class definition represents a commission employee.
3  #ifndef COMMISSION_H
4  #define COMMISSION_H
5
6  #include <string> // C++ standard string class
7  using std::string;
8
```

```
9  class CommissionEmployee
10 {
11 public:
12    CommissionEmployee( const string &, const string &, const string &,
13       double = 0.0, double = 0.0 );
14
15    void setFirstName( const string & ); // set first name
16    string getFirstName() const; // return first name
17
18    void setLastName( const string & ); // set last name
19    string getLastName() const; // return last name
20
21    void setSocialSecurityNumber( const string & ); // set SSN
22    string getSocialSecurityNumber() const; // return SSN
23
24    void setGrossSales( double ); // set gross sales amount
25    double getGrossSales() const; // return gross sales amount
26
27    void setCommissionRate( double ); // set commission rate
28    double getCommissionRate() const; // return commission rate
29
30    double earnings() const; // calculate earnings
31    void print() const; // print CommissionEmployee object
32 private:
33    string firstName;
34    string lastName;
35    string socialSecurityNumber;
36    double grossSales; // gross weekly sales
37    double commissionRate; // commission percentage
38 }; // end class CommissionEmployee
39
40 #endif
```

شکل ۱۳-۱ | فایل سرآیند کلاس CommissionEmployee.

```
1  // Fig. 13.2: CommissionEmployee.cpp
2  // Class CommissionEmployee member-function definitions.
3  #include <iostream>
4  using std::cout;
5
6  #include "CommissionEmployee.h" // CommissionEmployee class definition
7
8  // constructor
9  CommissionEmployee::CommissionEmployee(
10    const string &first, const string &last, const string &ssn,
11    double sales, double rate )
12    : firstName( first ), lastName( last ), socialSecurityNumber( ssn )
13 {
14    setGrossSales( sales ); // validate and store gross sales
15    setCommissionRate( rate ); // validate and store commission rate
16 } // end CommissionEmployee constructor
17
18 // set first name
19 void CommissionEmployee::setFirstName( const string &first )
20 {
21    firstName = first; // should validate
22 } // end function setFirstName
23
24 // return first name
25 string CommissionEmployee::getFirstName() const
26 {
27    return firstName;
28 } // end function getFirstName
29
30 // set last name
31 void CommissionEmployee::setLastName( const string &last )
32 {
33    lastName = last; // should validate
34 } // end function setLastName
35
36 // return last name
```

```
37 string CommissionEmployee::getLastName() const
38 {
39    return lastName;
40 } // end function getLastName
41
42 // set social security number
43 void CommissionEmployee::setSocialSecurityNumber( const string &ssn )
44 {
45    socialSecurityNumber = ssn; // should validate
46 } // end function setSocialSecurityNumber
47
48 // return social security number
49 string CommissionEmployee::getSocialSecurityNumber() const
50 {
51    return socialSecurityNumber;
52 } // end function getSocialSecurityNumber
53
54 // set gross sales amount
55 void CommissionEmployee::setGrossSales( double sales )
56 {
57    grossSales = ( sales < 0.0 ) ? 0.0 : sales;
58 } // end function setGrossSales
59
60 // return gross sales amount
61 double CommissionEmployee::getGrossSales() const
62 {
63    return grossSales;
64 } // end function getGrossSales
65
66 // set commission rate
67 void CommissionEmployee::setCommissionRate( double rate )
68 {
69    commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
70 } // end function setCommissionRate
71
72 // return commission rate
73 double CommissionEmployee::getCommissionRate() const
74 {
75    return commissionRate;
76 } // end function getCommissionRate
77
78 // calculate earnings
79 double CommissionEmployee::earnings() const
80 {
81    return getCommissionRate() * getGrossSales();
82 } // end function earnings
83
84 // print CommissionEmployee object
85 void CommissionEmployee::print() const
86 {
87    cout << "commission employee: "
88       << getFirstName() << ' ' << getLastName()
89       << "\nsocial security number: " << getSocialSecurityNumber()
90       << "\ngross sales: " << getGrossSales()
91       << "\ncommission rate: " << getCommissionRate();
92 } // end function print
```

شکل ۲-۱۳ | فایل پیاده‌سازی کلاس CommissionEmployee.

```
1  // Fig. 13.3: BasePlusCommissionEmployee.h
2  // BasePlusCommissionEmployee class derived from class
3  // CommissionEmployee.
4  #ifndef BASEPLUS_H
5  #define BASEPLUS_H
6
7  #include <string> // C++ standard string class
8  using std::string;
9
10 #include "CommissionEmployee.h" // CommissionEmployee class declaration
11
12 class BasePlusCommissionEmployee : public CommissionEmployee
```

```
13 {
14 public:
15    BasePlusCommissionEmployee( const string &, const string &,
16       const string &, double = 0.0, double = 0.0, double = 0.0 );
17
18    void setBaseSalary( double ); // set base salary
19    double getBaseSalary() const; // return base salary
20
21    double earnings() const; // calculate earnings
22    void print() const; // print BasePlusCommissionEmployee object
23 private:
24    double baseSalary; // base salary
25 }; // end class BasePlusCommissionEmployee
26
27 #endif
```

شکل ۳-۱۳ | فایل سرآیند کلاس BasePlusCommissionEmployee.

```
1  // Fig. 13.4: BasePlusCommissionEmployee.cpp
2  // Class BasePlusCommissionEmployee member-function definitions.
3  #include <iostream>
4  using std::cout;
5
6  // BasePlusCommissionEmployee class definition
7  #include "BasePlusCommissionEmployee.h"
8
9  // constructor
10 BasePlusCommissionEmployee::BasePlusCommissionEmployee(
11    const string &first, const string &last, const string &ssn,
12    double sales, double rate, double salary )
13    // explicitly call base-class constructor
14    : CommissionEmployee( first, last, ssn, sales, rate )
15 {
16    setBaseSalary( salary ); // validate and store base salary
17 } // end BasePlusCommissionEmployee constructor
18
19 // set base salary
20 void BasePlusCommissionEmployee::setBaseSalary( double salary )
21 {
22    baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
23 } // end function setBaseSalary
24
25 // return base salary
26 double BasePlusCommissionEmployee::getBaseSalary() const
27 {
28    return baseSalary;
29 } // end function getBaseSalary
30
31 // calculate earnings
32 double BasePlusCommissionEmployee::earnings() const
33 {
34    return getBaseSalary() + CommissionEmployee::earnings();
35 } // end function earnings
36
37 // print BasePlusCommissionEmployee object
38 void BasePlusCommissionEmployee::print() const
39 {
40    cout << "base-salaried ";
41
42    // invoke CommissionEmployee's print function
43    CommissionEmployee::print();
44
45    cout << "\nbase salary: " << getBaseSalary();
46 } // end function print
```

شکل ٤-۱۳ | فایل پیاده‌سازی کلاس BasePlusCommissionEmployee.

در شکل ۵-۱۳، خطوط 20-19 یک شی **CommissionEmployee** و در خط 23 یک اشاره‌گر به این شی و در خطوط 27-26 یک شی **BasePlusCommissionEmployee** و در خط 30 یک اشاره‌گر به

این شی ایجاد می‌شود. خطوط 37 و 39 از نام این شی‌ها برای احضار تابع عضو **print** هر یک از این شی‌ها استفاده می‌کنند. خط 42 آدرس کلاس مبنای شی **CommissionEmpolyee** را به اشاره‌گر کلاس مبنای **CommissionEmployeePtr** تخصیص می‌دهد، که خط 45 با استفاده از آن، تابع عضو **print** را برای شی **CommissionEmployee** احضار می‌نماید. با این عمل، نسخه **print** تعریف شده در کلاس مبنای **CommissionEmpolyee** احضار می‌شود. به همین ترتیب خط 48 آدرس شی کلاس مشتق شده **basePlusCommissionEmployee** را به اشاره‌گر کلاس مشتق شده **basePlusCommissionEmployeePtr** تخصیص می‌دهد، که خط 52 با استفاده از آن، تابع عضو print را بر روی شی **BasePlusCommissionEmployee** فراخوانی می‌کند. با این عمل، نسخه **print** تعریف شده در کلاس مشتق شده **BasePlusCommissionEmployee** فراخوانی می‌گردد. سپس خط **55** مبادرت به تخصیص آدرس شی کلاس مشتق شده **basePlusCommissionEmployee** به اشاره‌گر کلاس مبنا **CommissionEmployerPtr** می‌کند، که خط 59 با استفاده از آن تابع عضو **print** را احضار می‌نماید. کامپایلر C++ اجازه تغییر از یک حالت به حالت دیگر را می‌دهد، چرا که یک شی از یک کلاس مشتق شده، یک شی از کلاس مبنای خودش است (رابطه is-a). توجه کنید با وجود اینکه اشاره‌گر کلاس مبنا **CommissionEmployee** به یک کلاس مشتق شده **BasePlusCommissionEmployee** اشاره می‌کند، تابع عضو **print** کلاس مبنای **CommissionEmployee** احضار می‌شود (بجای تابع **print** کلاس **BasePlusCommissionEmployee**).

```
1  // Fig. 13.5: fig13_05.cpp
2  // Aiming base-class and derived-class pointers at base-class
3  // and derived-class objects, respectively.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7  using std::fixed;
8
9  #include <iomanip>
10 using std::setprecision;
11
12 // include class definitions
13 #include "CommissionEmployee.h"
14 #include "BasePlusCommissionEmployee.h"
15
16 int main()
17 {
18    // create base-class object
19    CommissionEmployee commissionEmployee(
20      "Sue", "Jones", "222-22-2222", 10000, .06 );
21
22    // create base-class pointer
23    CommissionEmployee *commissionEmployeePtr = 0;
24
25    // create derived-class object
26    BasePlusCommissionEmployee basePlusCommissionEmployee(
27       "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
28
29    // create derived-class pointer
30    BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;
```

```
31
32    // set floating-point output formatting
33    cout << fixed << setprecision( 2 );
34
35    // output objects commissionEmployee and basePlusCommissionEmployee
36    cout << "Print base-class and derived-class objects:\n\n";
37    commissionEmployee.print(); // invokes base-class print
38    cout << "\n\n";
39    basePlusCommissionEmployee.print(); // invokes derived-class print
40
41    // aim base-class pointer at base-class object and print
42    commissionEmployeePtr = &commissionEmployee; // perfectly natural
43    cout << "\n\n\nCalling print with base-class pointer to "
44       << "\nbase-class object invokes base-class print function:\n\n";
45    commissionEmployeePtr->print(); // invokes base-class print
46
47    // aim derived-class pointer at derived-class object and print
48    basePlusCommissionEmployeePtr = &basePlusCommissionEmployee; // natural
49    cout << "\n\n\nCalling print with derived-class pointer to "
50       << "\nderived-class object invokes derived-class "
51       << "print function:\n\n";
52   basePlusCommissionEmployeePtr->print();// invokes derived-class print
53
54    // aim base-class pointer at derived-class object and print
55    commissionEmployeePtr = &basePlusCommissionEmployee;
56    cout << "\n\n\nCalling print with base-class pointer to "
57       << "derived-class object\ninvokes base-class print "
58       << "function on that derived-class object:\n\n";
59    commissionEmployeePtr->print(); // invokes base-class print
60    cout << endl;
61    return 0;
62 } // end main
```

```
Print base-class and derived-class objects:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary:300.00

calling print with base-class pointer to
base-class object invokes base-class print function:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

calling print with derived-class pointer to
derived-class object invokes derived-class print function:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary:300.00

calling print with base-class pointer to derived-class object
invokes base-class print function on that derived-class object:

commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
```

شکل ۵-۱۳ | تخصیص آدرس شی‌های کلاس مبنا و کلاس مشتق شده به اشاره‌گرهای کلاس مبنا و کلاس مشتق شده.

خروجی هر تابع عضو **print** احضار شده در این برنامه آشکار می‌کند که فراخوانی یا احضار تابع بستگی به نوع دستگیر یا هندلی دارد (یعنی نوع اشاره‌گر یا مراجعه) که در فراخوانی تابع بکار گرفته شده است، نه به نوع شی که هندل به آن اشاره می‌کند. در بخش ۴-۳-۱۳، زمانیکه به معرفی توابع **virtual** پرداختیم، نشان خواهیم داد که می‌توان بجای توجه به نوع هندل، مبادرت به فراخوانی تابع کرد. شاهد خواهید بود که اینحالت در پیاده‌سازی رفتار چند ریختی بسیار تعیین کننده است و یکی از مباحث کلیدی این فصل نیز می‌باشد.

۲-۳-۱۳ هدایت اشاره‌گرهای کلاس مشتق شده بطرف شی‌های کلاس مشتق شده

در بخش ۱-۳-۱۳، اقدام به تخصیص آدرس یک شی از کلاس مشتق شده به اشاره‌گر یک کلاس مبنا کردیم و توضیح دادیم که کامپایلر C++ اجازه انجام چنین تخصیصی را می‌دهد، چرا که یک شی از کلاس مشتق شده یک شی از کلاس مبنا است (رابطه is-a). در شکل ۶-۱۳ یک رویه مخالف اخذ کرده‌ایم و اشاره‌گر کلاس مشتق شده را به طرف یک شی کلاس مبنا هدایت می‌کنیم. [نکته: این برنامه از کلاس **CommissinEmployee** و **BasePlusCommissionEmployee** شکل‌های ۱-۱۳ الی ۴-۱۳ استفاده کرده است.] خطوط 9-8 از شکل ۶-۱۳ یک شی **CommissionEmployee** ایجاد می‌کنند و خط 10 یک اشاره‌گر **BasePlusCommissionEmployee** ایجاد می‌نماید خط 14 مبادرت به تخصیص آدرس شی کلاس مبنا **CommissionEmployee** به اشاره‌گر کلاس مشتق شده **basePlusCommissionEmployeePtr** می‌کند، اما کامپایلر C++ خطا تولید می‌کند. کامپایلر مانع انجام چنین تخصیصی می‌شود، چرا که یک **CommissionEmployee** یک **BasePlusCommissionEmployee** نیست. اگر کامپایلر اجازه چنین تخصیصی را می‌داد، اتفاقات زیر بدنبال هم رخ می‌دادند. از طریق اشاره‌گر **BasePlusCommissionEmployee** می‌توانستیم هر تابع **BasePlusCommissionEmployee** شامل **setBaseSalary** را برای شی که اشاره‌گر بر آن اشاره دارد فراخوانی کنیم (یعنی شی کلاس مبنا **CommissionEmployee**). با این وجود، شی **CommissionEmployee** دارای تابع عضو **setBaseSalary** نیست و نمی‌تواند عضو داده **baseSalary** را تنظیم کند (چرا که این نوع کارمند دارای حقوق پایه نیست و دستمزد خود را براساس درصدی از میزان فروش دریافت می‌کند). پس اینکار می‌تواند منجر به مشکلاتی گردد، برای اینکه تابع عضو **setBaseSalary** فرض می‌کند که در اینجا یک عضو داده **baseSalary** برای تنظیم وجود دارد. این حافظه متعلق به شی **CommissinEmployee** نبوده و از اینرو تابع عضو **setBaseSalary** می‌تواند بر روی

اطلاعات با ارزش دیگری را که در حافظه قرار دارند، بازنویسی کند، در صورتیکه این اطلاعات متعلق به شی دیگری هستند.

```
1  // Fig. 13.6: fig13_06.cpp
2  // Aiming a derived-class pointer at a base-class object.
3  #include "CommissionEmployee.h"
4  #include "BasePlusCommissionEmployee.h"
5
6  int main()
7  {
8     CommissionEmployee commissionEmployee(
9        "Sue", "Jones", "222-22-2222", 10000, .06 );
10    BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;
11
12    // aim derived-class pointer at base-class object
13    // Error: a CommissionEmployee is not a BasePlusCommissionEmployee
14    basePlusCommissionEmployeePtr = &commissionEmployee;
15    return 0;
16 } // end main
```

Borland C++ command-line compiler error message:

```
Error E2034 Fig13_06\fig13_06.cpp 14: Cannot convert 'CommissionEmployee
*'
   to 'BasePluseCommissionEmployee *' in function main()
```

GNU C++ compiler error message:

```
fig13_06.cpp:14: error: invalid conversion from 'CommissionEmployee*' to
   'BasePlusCommisssionEmployee*'
```

Microsoft Visual C++.NET compiler error message:

```
C:\cpphtp5_examples\ch13\Fig13_06\fig13_06.cpp(14): error C2440:
   '*': cannot convert from 'CommissionEmployee *__w64 ' to
   'BasePlusCommissionEmployee *'
        Cast from base to derived requires dynamic_cast or static_cast
```

شکل ٦-١٣ | هدایت اشاره‌گر کلاس مشتق شده بطرف یک شی کلاس مبنا.

۳-۳-۱۳ فراخوانی تابع عضو کلاس مشتق شده از طریق اشاره‌گرهای کلاس مبنا

بدون حضور اشاره‌گر کلاس مبنا، کامپایلر اجازه می‌دهد تا فقط توابع عضو کلاس مبنا را فراخوانی کنیم. از اینرو اگر اشاره‌گر کلاس مبنا بطرف یک شی کلاس مشتق شده هدایت گردد، و مبادرت به دسترسی به یک عضو از کلاس مشتق شده شود، خطای زمان کامپایل رخ خواهد داد.

برنامه شکل ۷-۱۳ پیآمد مبادرت به احضار یک تابع عضو کلاس مشتق شده از طریق اشاره‌گر کلاس مبنا را نشان می‌دهد. [نکته: مجدداً از کلاس‌های **CommissionEmployee** و **BasePlusCommissionEmployee** شکل‌های ۱-۱۳ الی ۴-۱۳ استفاده کرده‌ایم]. خط 9 مبادرت به ایجاد **CommissionEployeePtr** می‌کند، که یک اشاره‌گر به یک شی **CommissionEmployee** است، و خطوط 11-10 یک شی **BasePlusCommissionEmployee** ایجاد می‌کند. خط 14 اشاره‌گر را بطرف شی مشتق شده از کلاس **BasePlusCommissionEmployee** هدایت می‌نماید. از بخش ۱-۳-۱۳ بخاطر دارید که کامپایلر C++ اجازه انجام چنین کاری را می‌دهد، چرا که یک **BasePlusCommissionEmployee** یک **CommissionEmployee** است.

```
1  // Fig. 13.7: fig13_07.cpp
2  // Attempting to invoke derived-class-only member functions
3  // through a base-class pointer.
4  #include "CommissionEmployee.h"
5  #include "BasePlusCommissionEmployee.h"
```

```
6
7  int main()
8  {
9     CommissionEmployee *commissionEmployeePtr = 0; // base class
10    BasePlusCommissionEmployee basePlusCommissionEmployee(
11       "Bob", "Lewis", "333-33-3333", 5000, .04, 300 ); // derived class
12
13    // aim base-class pointer at derived-class object
14    commissionEmployeePtr = &basePlusCommissionEmployee;
15
16    // invoke base-class member functions on derived-class
17    // object through base-class pointer
18    string firstName = commissionEmployeePtr->getFirstName();
19    string lastName = commissionEmployeePtr->getLastName();
20    string ssn = commissionEmployeePtr->getSocialSecurityNumber();
21    double grossSales = commissionEmployeePtr->getGrossSales();
22    double commissionRate = commissionEmployeePtr->getCommissionRate();
23
24    // attempt to invoke derived-class-only member functions
25    // on derived-class object through base-class pointer
26    double baseSalary = commissionEmployeePtr->getBaseSalary();
27    commissionEmployeePtr->setBaseSalary( 500 );
28    return 0;
29 } // end main
```

شکل ۷-۱۳ | اقدام به احضار فقط توابع کلاس مشتق شده از طریق اشاره‌گر کلاس مبنا.

خطوط 22-18 توابع عضو کلاس مبنا بنام‌های **getFirstName**، **getLastName**، **getSocialSecurityName**، **getGrossSales** و **getCommissionRate** از طریق اشاره‌گر کلاس مبنا فراخوانی می‌کنند. تمام این فراخوانی‌ها مشروع هستند، برای اینکه **BasePlusCommissionEmployee** این توابع عضو را از **CommissionEmployee** به ارث برده است. می‌دانیم که اشاره‌گر **CommissionEmployeePtr** بطرف یک شی **BasePlusCommissionEmployee** هدایت شده، از اینرو در خطوط 27-26 مبادرت به احضار توابع عضو **getBaseSalary** و **setBaseSalary** از **BasePlusCommissionEmployee** کرده‌ایم. کامپایلر C++ بر روی هر دو این خطوط خطا تولید می‌کند، به این دلیل که اینها توابع عضو کلاس **CommissionEmployee** نیستند.

۱۳-۳-٤ توابع virtual

در بخشی ۱-۳-۱۳ مبادرت به هدایت اشاره‌گر کلاس مبنا **CommissionEmployee** به طرف شی **BasePlusCommissionEmployee** از یک کلاس مشتق شده کردیم، سپس تابع عضو **print** را از طریق این اشاره‌گر فراخوانی نمودیم. بخاطر دارید که نوع دستگیره تعیین می‌کند که کدام تابع کلاس احضار شود. در این مورد، اشاره‌گر **CommissionEmployee** تابع عضو **print** متعلق به **CommissionEmployee** را بر روی شی **BasePlusCommissionEmployee** فراخوانی می‌کند، ولو اینکه اشاره‌گر به طرف شی **BasePlusCommissionEmployee** هدایت شده باشد که خود دارای تابع **print** خاص خودش است. به کمک توابع **virtual** (مجازی)، نوع شی اشاره شده و نه نوع دستگیره، تعیین می‌کند که کدام نسخه از یک تابع مجازی فراخوانی شود.

ابتدا به دلیل سودمند بودن توابع مجازی می‌پردازیم. فرض کنید که مجموعه‌ای از کلاس‌های شکل همانند **Circle** (دایره)، **Triangle** (مثلث)، **Rectangle** (مستطیل) و **Square** (مربع) داریم که همگی از کلاس مبنا **Shape** مشتق شده‌اند. هر کدامیک از کلاس‌ها می‌توانند از قابلیت ترسیم خود از طریق یک تابع عضو بنام **draw** برخوردار باشد. اگرچه هر کلاسی دارای تابع **draw** متعلق بخود است، عملکرد این تابع برای هر شکل با دیگری کاملاً متفاوت خواهد بود. در برنامه‌ای که مجموعه‌ای از شکل‌ها را ترسیم می‌کند، قابلیت تلقی کردن تمام شکل‌ها بصورت شی‌های از کلاس مبنا **Shape** سودمند خواهد بود. سپس، برای ترسیم هر شکلی، می‌توانیم به آسانی از یک اشاره‌گر **Shape** کلاس مبنا برای فراخوانی تابع **draw** استفاده کرده و به برنامه اجازه دهیم تا بصورت دینامیکی (یعنی در زمان اجرا) تعیین کند که کدام تابع **draw** کلاس مشتق شده برحسب نوع شی که اشاره‌گر **Shape** به آن اشاره می‌کند، بکار گرفته شود.

برای داشتن چنین رفتاری، ابتدا تابع **draw** را در کلاس مبنا بعنوان یک تابع **virtual** اعلان کرده و تابع **draw** در هر کلاس مشتق شده را برای ترسیم شکل مقتضی **override** می‌کنیم. از منظر پیاده‌سازی، **override** کردن یک تابع تفاوتی با تعریف مجدد آن ندارد (روشی که تا بدین جا از آن استفاده کرده‌ایم). یک تابع **override** شده در یک کلاس مشتق شده دارای همان امضاء و نوع برگشتی است (یعنی *نوع اولیه* یا *prototype*). اگر تابع کلاس مبنا را بصورت **virtual** اعلان نکنیم، می‌توانیم آن تابع را مجدداً تعریف کنیم. در مقابل اگر تابع کلاس مبنا را بصورت **virtual** اعلان کنیم، می‌توانیم آن تابع را **override** کرده تا از رفتار چند ریختی بهره‌مند گردیم.

می‌توانیم نمونه اولیه تابع فوق را با قرار دادن کلمه کلیدی **virtual** در کلاس مبنا، بصورت زیر بعنوان یک تابع **virtual** اعلان کنیم. برای مثال **virtual void draw() const** می‌تواند در کلاس مبنای **Shape** جای داده شود. در عبارت فوق تابع **draw** بصورت یک تابع **virtual** اعلان شده که هیچ آرگومانی دریافت نمی‌کند و چیزی هم برگشت نمی‌دهد. تابع بصورت **const** اعلان شده است چرا که تابع **draw** تغییری در شی **Shap** که برای ترسیم آن فراخوانی شده، بوجود نمی‌آورد. ضرورتاً توابع **virtual** مجبور نیستند تا بصورت **const** اعلان شوند.

اگر برنامه‌ای مبادرت به فراخوانی یک تابع **virtual** از طریق اشاره‌گر یک کلاس مبنا به یک شی از کلاس مشتق شده کند (مثلاً **ShapePtr->draw()**)، برنامه بصورت دینامیکی (یعنی در زمان اجرا) تابع صحیح **draw** را براساس نوع شی و نه نوع اشاره‌گر انتخاب خواهد کرد. انتخاب تابع مقتضی برای فراخوانی در زمان اجرا (بجای زمان کامپایل) بعنوان *مقیدسازی دینامیکی* (*dynamic binding*) یا *مقیدسازی تاخیری* (*late binding*) شناخته می‌شود.

زمانیکه یک تابع **virtual** توسط مراجعه‌ای به یک شی خاص توسط نام و استفاده از عملگر انتخاب عضو، نقطه (مثلاً **squarObject.draw()**) فراخوانی می‌شود، احضار تابع در زمان کامپایل مقرر می‌شود (که به اینحالت مقیدسازی استاتیک گفته می‌شود) و تابع **virtual** که فراخوانی شده یک تابع تعریف شده برای کلاسی از شی مشخص است، که این رفتار نشاندهنده چند ریختی نیست. از اینرو، مقیدسازی دینامیکی با توابع **virtual** فقط با دستگیره‌های اشاره‌گر (مراجعه) اتفاق می‌افتد.

حال اجازه دهید تا ببینیم چگونه توابع **virtual** می‌تواند نشاندهنده رفتار چند ریختی در سلسله مراتب کارمندی باشند. شکل‌های ۸-۱۳ و ۹-۱۳ فایلهای سرآیند برای کلاس‌های **CommissionEmployee** و **BasePlusCommissionEmployee** هستند. توجه کنید که تنها تفاوت موجود مابین این فایلها و آنهایی که در شکل‌های ۱-۱۳ و ۳-۱۳ قرار دارند در این است که توابع عضو **earnings** و **print** را بصورت **virtual** اعلان کرده‌ایم (خطوط 31-30 از شکل ۸-۱۳ و خطوط 22-21 از شکل ۹-۱۳). چون توابع **earnings** و **print** بصورت **virtual** در کلاس **CommissionEmployee** هستند، توابع **earnings** و **print** در **BasePlusCommissionEmployee** اقدام به **override** کردن کلاس **CommissionEmployee** می‌کنند. اکنون، اگر مبادرت به هدایت اشاره‌گر کلاس مبنای **CommissionEmployee** بطرف یک شی از کلاس مشتق شده **BasePlusCommissionEmployee** کنیم و برنامه از آن اشاره‌گر برای فراخوانی هر یک از دو تابع **earnings** یا **print** استفاده کند، تابع متناظر شی **BasePlusCommissionEmployee** فراخوانی خواهد شد. در اینجا هیچ تغییری در پیاده‌سازی تابع عضو از کلاس‌های **CommidssionEmployee** و **BasePlusCommissionEmployee** رخ نمی‌دهد، از اینرو استفاده مجددی از نسخه‌های شکل ۲-۱۳ و ۴-۱۳ می‌کنیم.

برای ایجاد برنامه شکل ۱۰-۱۳، تغییراتی در برنامه شکل ۵-۱۳ اعمال کرده‌ایم. خطوط 57-46 مجدداً نشان می‌دهند که می‌توان یک اشاره‌گر **CommissionEmployee** را بطرف یک شی **CommissionEmployee** هدایت کرده و توابع آنرا احضار کرد. اینحالت برای اشاره‌گر **BasePlusCommissionEmployee** نیز صادق است. در خط 60، اشاره‌گر کلاس مبنا **CommissionEmplyeePtr** بطرف شی کلاس مشتق شده **BasePlusCommissionEmployee** هدایت شده است. دقت کنید زمانیکه خط 67 مبادرت به احضار تابع عضو **print** از طریق اشاره‌گر کلاس مبنا می‌کند، تابع عضو **print** کلاس مشتق شده **BasePlusCommissionEmployee** احضار می‌گردد، از اینرو خط 67 متن متفاوتی از خط 59 در شکل ۵-۱۳ را در خروجی قرار می‌دهد (زمانیکه تابع عضو **print** بصورت **virtual** اعلان نشده بود).

```
1  // Fig. 13.8: CommissionEmployee.h
2  // CommissionEmployee class definition represents a commission employee.
3  #ifndef COMMISSION_H
```

```
4  #define COMMISSION_H
5
6  #include <string> // C++ standard string class
7  using std::string;
8
9  class CommissionEmployee
10 {
11 public:
12    CommissionEmployee( const string &, const string &, const string &,
13       double = 0.0, double = 0.0 );
14
15    void setFirstName( const string & ); // set first name
16    string getFirstName() const; // return first name
17
18    void setLastName( const string & ); // set last name
19    string getLastName() const; // return last name
20
21    void setSocialSecurityNumber( const string & ); // set SSN
22    string getSocialSecurityNumber() const; // return SSN
23
24    void setGrossSales( double ); // set gross sales amount
25    double getGrossSales() const; // return gross sales amount
26
27    void setCommissionRate( double ); // set commission rate
28    double getCommissionRate() const; // return commission rate
29
30    virtual double earnings() const; // calculate earnings
31    virtual void print() const; // print CommissionEmployee object
32 private:
33    string firstName;
34    string lastName;
35    string socialSecurityNumber;
36    double grossSales; // gross weekly sales
37    double commissionRate; // commission percentage
38 }; // end class CommissionEmployee
39
40 #endif
```

شکل ۸-۱۳ | فایل سرآیند کلاس CommissionEmployee که در آن توابع earnings و print بعنوان virtual اعلان شده‌اند.

```
1  // Fig. 13.9: BasePlusCommissionEmployee.h
2  // BasePlusCommissionEmployee class derived from class
3  // CommissionEmployee.
4  #ifndef BASEPLUS_H
5  #define BASEPLUS_H
6
7  #include <string> // C++ standard string class
8  using std::string;
9
10 #include "CommissionEmployee.h" // CommissionEmployee class declaration
11
12 class BasePlusCommissionEmployee : public CommissionEmployee
13 {
14 public:
15    BasePlusCommissionEmployee( const string &, const string &,
16       const string &, double = 0.0, double = 0.0, double = 0.0 );
17
18    void setBaseSalary( double ); // set base salary
19    double getBaseSalary() const; // return base salary
20
21    virtual double earnings() const; // calculate earnings
22    virtual void print() const; // print BasePlusCommissionEmployee object
23 private:
24    double baseSalary; // base salary
25 }; // end class BasePlusCommissionEmployee
26
27 #endif
```

شکل ۹-۱۳ | فایل سرآیند کلاس BasePlusCommissionEmployee که در آن توابع earnings و print بعنوان virtual اعلان شده‌اند.

مشاهده کردید که اعلان یک تابع عضو **virtual** سبب می‌شود تا برنامه بصورت دینامیکی تعیین کند که کدام تابع براساس نوع شی که دستگیره به آن اشاره می‌کند، فراخوانی گردد (بجای توجه به نوع دستگیره). تصمیم در مورد اینکه کدام تابع فراخوانی شود، مثالی از چند ریختی است.

مجدداً توجه کنید زمانیکه **CommissionEmployeePtr** به یک شی **CommissionEmployee** اشاره می‌کند (خط 46) تابع **print** کلاس **CommissionEmployee** احضار شده و زمانیکه **CommissionEmployePtr** به یک شی **BasePlusCommissionEmployee** اشاره می‌کند، تابع **print** کلاس **BasePlusCommissionEmployee** احضار می‌گردد. از اینرو، پیغام یکسان – در این مورد، **print** – به انواع مختلفی از شی‌ها ارسال می‌شود که رابطه ارث‌بری با کلاس مبنا دارند، و وارد فرم‌های متعدد نشده که نشاندهنده رفتار چند ریختی است.

```
1  // Fig. 13.10: fig13_10.cpp
2  // Introducing polymorphism, virtual functions and dynamic binding.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6  using std::fixed;
7
8  #include <iomanip>
9  using std::setprecision;
10
11 // include class definitions
12 #include "CommissionEmployee.h"
13 #include "BasePlusCommissionEmployee.h"
14
15 int main()
16 {
17    // create base-class object
18    CommissionEmployee commissionEmployee(
19       "Sue", "Jones", "222-22-2222", 10000, .06 );
20
21    // create base-class pointer
22    CommissionEmployee *commissionEmployeePtr = 0;
23
24    // create derived-class object
25    BasePlusCommissionEmployee basePlusCommissionEmployee(
26       "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
27
28    // create derived-class pointer
29    BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;
30
31    // set floating-point output formatting
32    cout << fixed << setprecision( 2 );
33
34    // output objects using static binding
35    cout << "Invoking print function on base-class and derived-class "
36       << "\nobjects with static binding\n\n";
37    commissionEmployee.print(); // static binding
38    cout << "\n\n";
39    basePlusCommissionEmployee.print(); // static binding
40
41    // output objects using dynamic binding
42    cout << "\n\n\nInvoking print function on base-class and "
43       << "derived-class \nobjects with dynamic binding";
44
```

```
45    // aim base-class pointer at base-class object and print
46    commissionEmployeePtr = &commissionEmployee;
47    cout << "\n\nCalling virtual function print with base-class pointer"
48       << "\nto base-class object invokes base-class "
49       << "print function:\n\n";
50    commissionEmployeePtr->print(); // invokes base-class print
51
52    // aim derived-class pointer at derived-class object and print
53    basePlusCommissionEmployeePtr = &basePlusCommissionEmployee;
54    cout << "\n\nCalling virtual function print with derived-class "
55       << "pointer\nto derived-class object invokes derived-class "
56       << "print function:\n\n";
57    basePlusCommissionEmployeePtr->print(); // invokes derived-class print
58
59    // aim base-class pointer at derived-class object and print
60    commissionEmployeePtr = &basePlusCommissionEmployee;
61    cout << "\n\nCalling virtual function print with base-class pointer"
62       << "\nto derived-class object invokes derived-class "
63       << "print function:\n\n";
64
65    // polymorphism; invokes BasePlusCommissionEmployee's print;
66    // base-class pointer to derived-class object
67    commissionEmployeePtr->print();
68    cout << endl;
69    return 0;
70 } // end main
```

```
Invoking print function on base-class and derived-class
Objects with static binding

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary:300.00

Invoking print function on base-class and derived-class
Objects with dynamic binding

calling virtual function print with base-class pointer to
base-class object invokes base-class print function:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

calling virtual function print with derived-class pointer to
derived-class object invokes derived-class print function:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary:300.00

calling virtual function print with base-class pointer to derived-class
object invokes derived-class print function:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00
```

شکل ۱۰-۱۳ | توصیف چند ریختی با فراخوانی یک تابع virtual کلاس مشتق شده از طریق اشاره‌گر کلاس مبنا به یک کلاس مشتق شده.

۵-۳-۱۳ تخصیص‌های قابل انجام مابین شی‌ها و اشاره‌گرهای کلاس مبنا و کلاس مشتق شده

اکنون که شاهد یک برنامه کامل که مبادرت به پردازش شی‌های مشتق شده به روش چند ریختی بودید، بطور خلاصه مواردی را که می‌توانید با شی‌های کلاس مشتق شده، مبنا و اشاره‌گر انجام دهید و آنهایی که نمی‌توانید انجام دهید، مطرح می‌کنیم.

اگرچه یک شی کلاس مشتق شده یک شی از یک کلاس مبنا است، اما این دو شی با هم تفاوت‌های دارند. همانطوری که قبلاً هم بحث شدهٔ، شی‌های کلاس مشتق شده در صورتیکه شی‌های از کلاس مبنا باشند می‌توانند بصورت کلاس مبنا در نظر گرفته شوند. این یک رابطه منطقی است، چرا که کلاس مشتق شده حاوی تمام اعضا از کلاس مبنا است. با این وجود نمی‌توان با شی‌های کلاس مبنا مثل اینکه شی‌های از کلاس مشتق شده هستند رفتار کرد. به همین دلیل، هدایت اشاره‌گر کلاس مشتق شده بطرف یک شی کلاس مبنا، بدون یک تبدیل صریح امکان‌پذیر نمی‌باشد. عمل تبدیل به کامپایلر کمک می‌کند تا از صدور پیغام خطا اجتناب کند. در چنین حالتی، با استفاده از تبدیل می‌گوید که «من از خطری کاری که انجام می‌دهم مطلع هستم و مسئولیت تمام کارها را برعهده می‌گیرم.»

در بخش جاری و فصل دوازدهم، به بررسی چهار روش در هدایت اشاره‌گرهای کلاس مبنا و اشاره‌گرهای کلاس مشتق شده بطرف شی‌های کلاس مبنا و کلاس مشتق شده پرداختیم:

۱- هدایت اشاره‌گر کلاس مبنا بطرف شی از کلاس مبنا کار سرراستی است. با این عمل اشاره‌گر مبادرت به فراخوانی کلاس مبنا می‌کند.

۲- هدایت اشاره‌گر کلاس مشتق شده بطرف شی از کلاس مشتق شده کار سرراستی است. با این عمل اشاره‌گر مبادرت به فراخوانی کلاس مشتق شده می‌کند.

۳- هدایت اشاره‌گر یک کلاس مبنا بطرف یک شی از کلاس مشتق شده، خطری ندارد، چرا که شی از کلاس مشتق شده یک شی از کلاس مبنای خودش است. با این وجود، از این اشاره‌گر می‌توان فقط در احضار توابع عضو کلاس مبنا استفاده کرد. اگر برنامه‌نویس از طریق اشاره‌گر کلاس مبنا مبادرت به اشاره به یک عضو فقط کلاس مشتق شده کند، کامپایلر خطا گزارش خواهد کرد.

برای اجتناب از این خطا، برنامه‌نویس بایستی اشاره‌گر کلاس مبنا را تبدیل به اشاره‌گر کلاس مشتق شده نماید. سپس می‌توان از اشاره‌گر کلاس مشتق شده برای فراخوانی کل توابع شی از کلاس مشتق شده استفاده کرد. با این همه این روش خطراتی نیز دارد که بخش ۸-۱۳ به بررسی و رفع آن خواهیم پرداخت.

۴- هدایت اشاره‌گر کلاس مشتق شده بطرف شی از کلاس مبنا، خطای کامپایل تولید می‌کند. رابطه is-a (است–یک) فقط بر روی یک کلاس مشتق شده بصورت مستقیم یا غیرمستقیم از کلاس مبنا خود معتبر است و عکس آن صادق نیست. یک شی از کلاس مبنا حاوی عضوهای خاص کلاس مشتق شده نیست که بتواند اشاره‌گر کلاس مشتق شده را احضار نماید.

۱۳-٤ عبارت switch

یکی از روش‌های تعیین نوع یک شی در یک برنامه بزرگ استفاده از عبارت **switch** است. این عبارت امکان می‌دهد تا مابین انواع شی‌ها تفاوت قائل شده، سپس عمل مقتضی را بر روی آن شی مشخص انجام دهیم. برای مثال در سلسله مراتب شکل‌ها که در آن هر شی دارای صفت **shaptType** است، یک عبارت **switch** می‌تواند به بررسی **shapeType** شی پرداخته و تعیین کند که کدام تابع **print** فراخوانی گردد.
با این همه، استفاده از **switch** سبب می‌شود تا منطق برنامه در معرض دید قرار گرفته و مهیا برای مشکلات شود. برای مثال، امکان دارد برنامه‌نویس انجام تستی بر روی نوع خاصی را یا قرار دادن تمام حالات ممکنه را در عبارت **switch** را فراموش کند. به هنگام اصلاح یک سیستم مبتنی بر **switch** که با افزودن نوع‌های جدید همراه است، امکان دارد برنامه‌نویس وارد کردن حالات جدید را در تمام عبارات **switch** وابسته فراموش نماید. هر افزودن یا حذف کلاسی مستلزم اصلاح در کلیه عبارات **switch** است، بررسی چنین عباراتی می‌تواند زمانبر بوده و مستعد و زمینه‌ساز خطا است.

۱۳-۵ کلاس‌های انتزاعی و توابع virtual محض

هنگامی که در مورد نوع یک کلاس فکر می‌کنیم، فرض ما بر این است که برنامه‌ها اقدام به ایجاد شیی‌ها از نوع تعیین شده خواهند کرد. با این همه، گاهی اوقات فقط کلاس‌ها تعریف می‌شوند و برنامه‌نویسان هرگز قصد ندارند هیچ شیی را نمونه‌سازی کنند. چنین کلاس‌هایی، کلاس‌های *انتزاعی* نامیده می‌شوند. چرا که چنین کلاس‌هایی معمولاً بعنوان کلاس‌های مبنا در سلسله مراتب توارث بکار گرفته می‌شوند، از اینرو معمولاً با عنوان کلاس‌های *مبنای انتزاعی* شناخته می‌شوند. این کلاس‌ها نمی‌توانند برای نمونه‌سازی شیی‌ها بکار گرفته شوند. کلاس‌های انتزاعی کامل نیستند. کلاس‌های مشتق شده بایستی بعنوان بخش‌های مفقود شده تعریف شوند.

منظور از یک کلاس انتزاعی فراهم آوردن یک کلاس مبنای مقتضی از سایر کلاس‌ها است که ممکن است به ارث برسند. کلاس‌های که از چنین شی‌هایی نمونه‌سازی می‌شوند، کلاس‌های *مقید* نام دارند. چنین کلاس‌هایی تدارک بیننده هر تابع عضو هستند که تعریف شده‌اند. برای مثال می‌توانیم یک کلاس مبنای انتزاعی بنام **TwoDimensionalObject** و کلاس‌های مشتق شده مقید همانند **Squar**،

Circle و **Triangle** داشته باشیم. همچنین می‌توانیم یک کلاس مبنای انتزاعی بنام **ThreeDimensionalObject** و کلاس‌های مشتق شده مقید همانند **Cube**، **Sphere** و **Cylinder** داشته باشیم. کلاس‌های مبنای انتزاعی برای تعریف شیی‌های واقعی بسیار کلی هستند، از اینرو قبل از اینکه شیی را نمونه‌سازی کنیم باید با دقت در مورد آن فکر کنیم. برای مثال، اگر شخصی به شما بگوید "شکلی دو بعدی ترسیم کنید"، باید پرسید چه شکلی؟

یک سلسله مراتب توارث نیازی به کلاس‌های انتزاعی ندارد، اما همانطوری که مشاهده خواهید کرد، بسیاری از سیستم‌های خوب شی‌گرا، دارای سلسله مراتب کلاسی مناسب با کلاس‌های مبنا انتزاعی هستند. در برخی از موارد کلاس‌های انتزاعی، چند سطح فوقانی را در سلسله مراتب تشکیل می‌دهند. یک مثال خوب در این زمینه، سلسله مراتب شکل‌ها در شکل ۳-۱۲ است که با کلاس مبنای انتزاعی **Shape** شروع می‌شود. در سطح بعدی سلسله مراتب دو کلاس مبنای انتزاعی دیگر بنام‌های **TwoDimensionalShape** (شکل‌های دوبعدی) و **ThreeDimensinalShape** (شکل‌های سه بعدی) داریم. سطح بعدی سلسله مراتب کلاس‌های غیرانتزاعی برای شکل‌های دوبعدی را تعریف کرده است (بنام‌های **Circle**، **Square** و **Triangle**) و برای شکل‌های سه بعدی بنام‌های **Sphere**، **Cube** و **Tetrahedron** (چهارسطحی).

با اعلان یک یا چند تابع **virtual** یک کلاس بصورت محض، آن کلاس بصورت یک کلاس انتزاعی ایجاد می‌شود. با قرار دادن "**0** =" در اعلان یک تابع **virtual** آن تابع بصورت یک تابع **virtual** محض مشخص می شود، بصورت

```
virtual void draw() const = 0; //pure virtual function
```

"**0**=" بعنوان تصریح کننده محض شناخته می‌شود. توابع **virtual** محض دارای پیاده‌سازی نیستند. هر کلاس غیرانتزاعی بایستی تمام توابع **virtual** محض کلاس مبنا را **override** کند با پیاده‌سازی غیرانتزاعی توابع آنها. تفاوت موجود مابین یک تابع **virtual** و یک تابع **virutal** محض در این است که تابع **virtual** دارای پیاده‌سازی بوده و به کلاس مشتق شده گزینه‌ای برای **override** (لغو کردن) تابع اعطا می‌کند، در مقابل، یک تابع **virtual** محض دارای پیاده‌سازی نبوده و کلاس مشتق شده را ملزم به **oveeride** کردن تابع می‌نماید (به همین دلیل است که کلاس مشتق شده غیرانتزاعی می‌شود، در غیر اینصورت کلاس مشتق شده، انتزاعی باقی می‌ماند).

از توابع **virtual** محض زمانی استفاده می‌شود که احساس شود کلاس مبنا نیازی به پیاده‌سازی یک تابع ندارد، اما برنامه‌نویس مایل است تمام کلاس‌های مشتق شده غیرانتزاعی را در پیاده‌سازی تابع داشته باشد. اگر به مثال فضایی خود در ابتدای فصل باز گردیم، متوجه می‌شوید که کلاس مبنا **SpaceObject** دارای

پیاده‌سازی برای تابع **draw** نبود. مثالی از یک تابع که می‌تواند بعنوان یک تابع **virtual** (و نه یک **virtual** محض) تعریف شود آن است که نامی برای شی برگشت دهد.

اگرچه نمی‌توانیم نمونه‌های از شی‌های یک کلاس مبنا انتزاعی ایجاد کنیم، اما می‌توانیم از کلاس مبنای انتزاعی بمنظور اعلان اشاره‌گرها و مراجعه‌های که می‌توانند به شی‌های از هر کلاس غیرانتزاعی مشتق شده از کلاس‌های انتزاعی مراجعه کند، ایجاد کنیم. معمولاً برنامه‌ها از چنین اشاره‌گر و مراجعه‌های برای کار با شی‌های کلاس مشتق شده به روش چندریختی استفاده می‌کنند.

چند ریختی نقش ویژه‌ای در پیاده‌سازی لایه‌های مختلف در سیستم‌های نرم‌افزاری دارد. برای مثال، در سیستم‌های عامل، هر نوع، دستگاه فیزیکی می‌تواند بطور کاملاً متفاوتی در کنار دستگاه‌های دیگر بکار بپردازد. حتی دستورات خواندن و نوشتن داده‌ها از دستگاه‌ها می‌توانند بطور کلی با یکدیگر متفاوت باشند. اجازه دهید تا به بررسی کاربرد دیگری از چند ریختی بپردازیم. مدیر صفحه نیاز به نمایش انواع مختلفی از شی‌ها شامل انواع شی‌های جدیدی دارد که برنامه‌نویس پس از نوشتن مدیر صحنه به آن اضافه خواهد کرد. همچنین سیستم می‌تواند نیازمند به نمایش انواع مختلفی از شکل‌ها همانند دایره‌ها، مثلث‌ها یا مستطیل‌ها شود که از کلاس مبنای انتزاعی **Shape** مشتق شده‌اند. مدیر صحنه از اشاره‌گرهای **Shape** برای مدیریت شی‌های که به نمایش در می‌آیند استفاده می‌کند. برای ترسیم هر شی (صرفنظر از سطحی که کلاس شی در سلسله مراتب توارث قرار دارد)، مدیر صحنه از یک اشاره‌گر کلاس مبنا استفاده می‌کند تا تابع **draw** برای آن شی را فراخوانی کند، که یک تابع **virtual** محض در کلاس مبنای **Shape** است، بنابر این هر کلاس غیرانتزاعی مشتق شده باید تابع **draw** را پیاده‌سازی کند. هر شی **Shape** در سلسله مراتب توارث از چگونگی ترسیم خود مطلع است. نیازی نیست که مدیر صحنه نگران نوع هر شی بوده یا اینکه نگران آن باشد که قبلاً با آن شی مواجه شده است یا نه.

غالباً در برنامه‌نویسی شیءگرا، یک کلاس *تکرار شونده* (iterator) تعریف می‌کنند که می‌تواند در میان تمام شیءهای موجود در یک حامل (همانند یک آرایه) حرکت کند. برای مثال، برنامه می‌تواند لیستی از شیءهای موجود در یک لیست پیوندی را با ایجاد یک شیء تکرار شونده به چاپ در آورده و سپس با فراخوانی مجدد تکرار شونده، به عنصر بعدی در لیست دست یابد. معمولاً از تکرار شونده‌ها در برنامه‌نویسی پولی‌مورفیک برای پیمایش یک آرایه یا لیست پیوندی از سطح‌های مختلف یک سلسله مراتب استفاده می‌شود. اشاره‌گرها در چنین لیستی تماماً اشاره‌گرهای کلاس مبنا هستند. برای مثال، لیستی از شیءهایی کلاس مبنا **TwoDimensionalShape** می‌تواند، حاوی شیءهایی از کلاس‌های **Square**،

Circle، **Triangle** و غیره باشد. با استفاده از پلی‌مورفیزم یک پیغام **draw** به هر شیی در لیست ارسال می‌شود و آن شیی بدرستی بر روی صفحه ترسیم می‌گردد.

۱۳-٦ مبحث آموزشی: سیستم پرداخت حقوق با استفاده از چند ریختی

در این بخش به بررسی مجدد سلسله مراتب **CommissionEmployee-BasePlusCommissionEmployee** که در بخش ۴-۱۲ به معرفی آن اقدام کردیم، می‌پردازیم. در این مثال، از کلاس انتزاعی و چند ریختی برای انجام محاسبات پرداخت حقوق برحسب نوع کارمند استفاده می‌کنیم. سلسله مراتب کارمندی که در این بخش ایجاد می‌کنیم قادر به حل مسئله زیر است:

شرکتی به کارمندان خود بطور هفتگی حقوق پرداخت می‌کند. کارمندان به چهار دسته تقسیم شده‌اند: کارمندانی که یک حقوق ثابت صرفنظر از ساعات کاری در هفته دریافت می‌کنند، کارمندانی که براساس ساعت کاری و اضافه کاری در طول هفته مازاد بر 40 ساعت حقوق دریافت می‌کنند، کارمندانی که براساس فروش حقوق دریافت می‌کنند و کارمندانی که علاوه بر حقوق ثابت درصدی از فروش نیز کمیسیون به آنها تعلق می‌گیرد. شرکت تصمیم دارد که تا پرداخت حقوق‌های جاری به کارمندانی که حقوق پایه همراه با کمیسیون از فروش دریافت می‌کنند، ۱۰ درصد به میزان فروش آنها پاداش اضافه نماید. شرکت مایل به پیاده‌سازی یک برنامه‌ای است تا محاسبات پرداخت حقوق را به روش چند ریختی انجام دهد.

از کلاس انتزاعی **Employee** برای عرضه مفهوم کلی یک کارمند استفاده می‌کنیم. کلاس‌های که مستقیماً از **Employee** مشتق می‌شوند عبارتند از **SalariedEmployee**، **CommissionEmployee** و **HourlyEmployee**. کلاس **BasePlusCommissionEmployee** از کلاس **CommissionEmploye** مشتق شده و نشاندهنده آخرین نوع کارمند است. دیاگرام UML این کلاس در شکل ۱۱-۱۳ بنمایش درآمده و سلسله مراتب توارث را برای برنامه پرداخت حقوق به روش چندریختی را نشان می‌دهد. دقت کنید که نام کلاس **Employee** بصورت ایتالیک (کج) نوشته شده که قراردادی در UML می‌باشد.

شکل ۱۱-۳ | دیاگرام UML کلاس سلسله مراتب Employee.

کلاس مبنای انتزاعی **Employee** اعلان کننده یک «*واسط*» یا "*interface*" برای سلسله مراتب است، که مجموعه‌ای از توابع عضو می‌باشد که برنامه می‌تواند بر روی تمام شیءهای **Employee** فراخوانی کند. هر کارمندی صرفنظر از روش محاسبه حقوق وی، دارای نام، نام خانوادگی و شماره تامین اجتماعی بوده از اینرو اعضای داده خصوصی عبارتند از: **firstName**، **lastName** و **socialSecurityNumber** که در کلاس مبنای انتزاعی **Employee** ظاهر می‌شوند.

در بخش‌های زیر اقدام به پیاده‌سازی سلسله مراتب کلاس **Employee** خواهیم کرد. در پنج بخش اول یک کلاس انتزاعی یا غیرانتزاعی ایجاد می‌کنیم. در بخش پایانی یک برنامه تست پیاده‌سازی می‌نمائیم که شیءهای از تمام این کلاس‌ها ایجاد کرده و آنها را به روش چند ریختی پردازش می‌کند.

۱-٦-۱۳ ایجاد کلاس مبنای انتزاعی Employee

کلاس **Employee** (شکل‌های ۱۳-۱۳ و ۱۴-۱۳ که در بخش‌های بعدی توضیح داده خواهند شد) توابع **earnings** و **print** به همراه توابع متعدد *get* و *set* که اعضای داده **Employee** را فراهم می‌آورند، تدارک دیده است. تابع **earnings** بطور مشخصی بر روی تمام کارمندان اعمال می‌شود. اما محاسبه هر حقوق بستگی به کلاس کارمند دارد. از اینرو **earnings** را بصورت **virtual** محض در کلاس مبنای **employee** اعلان کرده‌ایم چرا که در پیاده‌سازی پیش‌فرض راضی کننده نیست یعنی اطلاعات کافی برای تعیین میزان حقوق پرداختی که باید برگشت داده شود وجود ندارد. هر کلاس مشتق شده‌ای تابع **earnings** را با پیاده‌سازی مقتضی بکار می‌گیرد. برای محاسبه حقوق یک کارمند برنامه آدرس شی کارمند را به اشاره‌گر کلاس مبنا تخصیص می‌دهد، سپس تابع **earnings** بر روی آن شی فراخوانی می‌گردد.

برای نگهداری اشاره‌گرهای **Employee** از یک **vector** استفاده کرده‌ایم که هر کدام به یک شی **Employee** اشاره کند (البته، آنها نمی‌توانند شی‌های **Employee** باشند چرا که **Employee** یک کلاس انتزاعی است، با این وجد بدلیل توارث، هر شی از تمام کلاس‌های مشتق شده از **Employee** شی‌های از آن محسوب می‌شوند). برنامه در میان **vector** حرکت کرده وتابع **earnings** را برای هر شی کارمند فراخوانی می‌کند. C++ این فراخوانی‌های تابع را به روش چند ریختی پردازش می‌کند. با توجه به اینکه **earnings** بصورت یک تابع **virtual** محض در **Employee** اعلان شده، هر کلاس که مستقیماً از **Employee** مشتق شود که می‌خواهد بصورت یک کلاس غیرانتزاعی باشد سبب **override** (انتخاب تابع مناسب) شدن **earnings** می‌شود. با اینکار طراح کلاس سلسله مراتب خواهد توانست برای هر کلاس مشتق شده محاسبه مقتضی را داشته باشد، در صورتیکه کلاس مشتق شده براستی غیرانتزاعی باشد.

تابع **print** در کلاس **Employee** مبادرت به نمایش نام، نام خانوادگی و شماره تامین اجتماعی کارمند می‌کند. همانطوری که مشاهده خواهید کرد، هر کلاس مشتق شده از **Employee** تابع مناسب **print** را انتخاب کرده و نوع کارمند را در خروجی چاپ می‌کند (مانند "salaried employee:") و به دنبال آن مابقی اطلاعات کارمند را قرار می‌دهد.

دیاگرام شکل ۱۲-۱۳ نمایشی از پنج کلاس موجود در سلسله مراتب است که در سمت چپ آن و توابع **earnings** و **print** در سرستون‌ها قرار گرفته‌اند. برای هر کلاس، دیاگرام نتیجه دلخواه هر تابع را نشان می‌دهد. دقت کنید که کلاس **Employee** با **"=0"** برای تابع **earnings** همراه شده و نشان می‌دهد که این تابع یک تابع **virtual** محض است. هر کلاس مشتق شده تابع مناسب خود را برای انجام مقاصد مقتضی انتخاب می‌کند (یعنی تابع **earnings** را **override** می‌کند). در این دیاگرام توابع **get** و **set**

کلاس مبنای **Employee** را لیست نکرده‌ایم، چرا که آنها هیچ یک از توابع در کلاس‌های مشتق شده را **override** نمی‌کنند.

اجازه دهید تا به بررسی فایل سرآیند **Employee** (شکل ۱۳-۱۳) بپردازیم. توابع عضو **public** شامل یک سازنده (که نام، نام خانوادگی و شماره تامین اجتماعی را بعنوان آرگومان دریافت کرده (خط 12))، توابع *set* (که تنظیم کننده نام، نام خانوادگی و شماره تامین اجتماعی است (خطوط 17,14 و 20))، توابع *get* (که نام، نام خانوادگی و شماره تامین اجتماعی را برگشت می‌دهند (خطوط 18,15 و 21))، تابع **virtual** محض **earnings** (خط 21) و تابع **print** که **virtual** است، می‌باشند (خط 25).

بخاطر دارید که تابع **earnings** را بصورت یک تابع **virtual** محض اعلان کرده‌ایم، چرا که بایستی ابتدا از نوع کارمند مطلع شویم تا بتوانیم محاسبه مناسب برای حقوق آن نوع کارمند را تعیین نمائیم. اعلان این تابع بعنوان **virtual** محض بر این نکته دلالت دارد که هر کلاس مشتق شده غیرانتزاعی بایستی یک پیاده‌سازی مقتضی از **earnings** تدارک دیده و برنامه بتواند از اشاره‌گرهای **Employee** برای فراخوانی تابع **earnings** به روش چند ریختی برای هر نوع کارمند استفاده کند.

شکل ۱۴-۱۳ حاوی پیاده‌سازی تابع عضو برای کلاس **Employee** است. هیچ پیاده‌سازی برای تابع **earnings** که **virtual** است تدارک دیده نشده است. به سازنده **Employee** (خطوط 15-10) توجه کنید که مبادرت به اعتبارسنجی شماره تامین اجتماعی نمی‌کند. معمولاً بایستی چنین اعتبارسنجی در نظر گرفته شود.

	earnings	**print**
Employee	**=0**	*firstName lastName* **social security number:** *SSN*
Salaried-Employee	**weeklySalary**	**salaried employee:** *firstName lastName* **social security number:** *SSN* **weekly salary:** *weeklysalary*
Hourly-Employee	*if hours <= 40* **wage * hours** *if hours > 40* **(40 * wage) +** **((hours – 40)** *** wage * 1.5)**	**hourly employee:** *firstName lastName* **social security number:** *SSN* **hourly wage:** *wage*; **hours worked:** *hours*
Commission-Employee	**commissionRate * grossSales**	**commission employee:** *firstName lastName* **social security number:** *SSN* **gross sales:** *grossSales*; **commission rate:** *commissionRate*
BasePlus-Commission-Employee	**baseSalary +** **(commissionRate * grossSales)**	**base salaried commission employee:** *firstName lastName* **social security number:** *SSN* **gross sales:** *grossSales*; **commission rate:** *commissionRate;*

	base salary: *baseSalary*

شکل ۱۲-۱۳ | واسط چند ریختی برای سلسله مراتب کلاس‌های Employee.

```
1  // Fig. 13.13: Employee.h
2  // Employee abstract base class.
3  #ifndef EMPLOYEE_H
4  #define EMPLOYEE_H
5
6  #include <string> // C++ standard string class
7  using std::string;
8
9  class Employee
10 {
11 public:
12    Employee( const string &, const string &, const string & );
13
14    void setFirstName( const string & ); // set first name
15    string getFirstName() const; // return first name
16
17    void setLastName( const string & ); // set last name
18    string getLastName() const; // return last name
19
20    void setSocialSecurityNumber( const string & ); // set SSN
21    string getSocialSecurityNumber() const; // return SSN
22
23    // pure virtual function makes Employee abstract base class
24    virtual double earnings() const = 0; // pure virtual
25    virtual void print() const; // virtual
26 private:
27    string firstName;
28    string lastName;
29    string socialSecurityNumber;
30 }; // end class Employee
31
32 #endif // EMPLOYEE_H
```

شکل ۱۳-۱۳| فایل سرآیند کلاس Employee.

```
1  // Fig. 13.14: Employee.cpp
2  // Abstract-base-class Employee member-function definitions.
3  // Note: No definitions are given for pure virtual functions.
4  #include <iostream>
5  using std::cout;
6
7  #include "Employee.h" // Employee class definition
8
9  // constructor
10 Employee::Employee( const string &first, const string &last,
11    const string &ssn )
12    : firstName( first ), lastName( last ), socialSecurityNumber( ssn )
13 {
14    // empty body
15 } // end Employee constructor
16
17 // set first name
18 void Employee::setFirstName( const string &first )
19 {
20    firstName = first;
21 } // end function setFirstName
22
23 // return first name
24 string Employee::getFirstName() const
25 {
26    return firstName;
27 } // end function getFirstName
28
29 // set last name
30 void Employee::setLastName( const string &last )
31 {
32    lastName = last;
33 } // end function setLastName
```

```
34
35 // return last name
36 string Employee::getLastName() const
37 {
38    return lastName;
39 } // end function getLastName
40
41 // set social security number
42 void Employee::setSocialSecurityNumber( const string &ssn )
43 {
44    socialSecurityNumber = ssn; // should validate
45 } // end function setSocialSecurityNumber
46
47 // return social security number
48 string Employee::getSocialSecurityNumber() const
49 {
50    return socialSecurityNumber;
51 } // end function getSocialSecurityNumber
52
53 // print Employee's information (virtual, but not pure virtual)
54 void Employee::print() const
55 {
56    cout << getFirstName() << ' ' << getLastName()
57       << "\nsocial security number: " << getSocialSecurityNumber();
58 } // end function print
```

شکل ١٤-١٣| فایل پیاده‌سازی کلاس Employee.

به تابع **print** که **virtual** است (شکل ۱۴-۱۳، خطوط 58-54) توجه نمائید که دارای پیاده‌سازی بوده و در هر کلاس مشتق شده override می‌شود (یعنی به کنار گذاشته شده و تابع متناسب برای آن کلاس انتخاب و اجرا می‌گردد). با این همه، هر یک از این توابع از **print** نسخه کلاس انتزاعی برای چاپ اطلاعات مشترک در تمام کلاس‌های موجود در سلسله مراتب **Employee** استفاده می‌کنند.

۲-٦-۱۳ ایجاد کلاس مشتق شده غیرانتزاعی SalariedEmployee

کلاس **SalariedEmployee** (شکل‌های ۱۵-۱۳ و ۱۶-۱۳) از کلاس **Employee** مشتق شده است (خط 8 از شکل ۱۵-۱۳). توابع عضو سراسری (public) شامل سازنده‌ای هستند که نام، نام خانوادگی، شماره تامین اجتماعی و حقوق هفتگی را بعنوان آرگومان می‌پذیرد (خطوط 12-11)، یک تابع *set* برای تخصیص مقادیر جدید غیرمنفی به عضو داده **weeklySalary** (خط 14)، یک تابع *get* برای برگشت دادن مقدار **weeklySalary** (خط 15)، یک تابع **earnings** که **virtual** بوده و حقوق کارمندی از نوع **SalariedEmployee** را محاسبه می‌کند (خط 18) و یک تابع **print** که **virtual** بوده که نوع کارمند را یعنی: "salaried employee" و بدنبال آن اطلاعات خاص آن کارمند را که توسط تابع **print** کلاس مبنای **Employee** و تابع **getWeeklySalary** کلاس **SalariedEmployee** تهیه شده است، چاپ می‌کند.

برنامه شکل ۱۶-۱۳ حاوی پیاده‌سازی تابع عضو برای **SalariedEmployee** است. سازنده کلاس مبادرت به ارسال نام، نام خانوادگی و شماره تامین اجتماعی به سازنده **Employee** می‌کند (خط 11) تا اعضای داده **private** را که از کلاس مبنا به ارث برده شده، اما در دسترس کلاس مشتق شده نمی‌باشند،

مقداردهی اولیه شوند. تابع **earnings** در خطوط 33-30 مبادرت به توقف تابع **earnings** در **Employee** می‌کند که **virtual** محض است تا پیاده‌سازی غیرانتزاعی تدارک دیده شده برای **SalariedEmployee** حقوق هفتگی را برگشت دهد. اگر **earnings** را پیاده‌سازی نمی‌کردیم، کلاس **SalariedEmployee** می‌خواست که یک کلاس انتزاعی باشد و نتیجه هر عملی برای نمونه‌سازی یک شی از کلاس، خطای کامپایل بود.

```
1  // Fig. 13.15: SalariedEmployee.h
2  // SalariedEmployee class derived from Employee.
3  #ifndef SALARIED_H
4  #define SALARIED_H
5
6  #include "Employee.h" // Employee class definition
7
8  class SalariedEmployee : public Employee
9  {
10 public:
11    SalariedEmployee( const string &, const string &,
12       const string &, double = 0.0 );
13
14    void setWeeklySalary( double ); // set weekly salary
15    double getWeeklySalary() const; // return weekly salary
16
17    // keyword virtual signals intent to override
18    virtual double earnings() const; // calculate earnings
19    virtual void print() const; // print SalariedEmployee object
20 private:
21    double weeklySalary; // salary per week
22 }; // end class SalariedEmployee
23
24 #endif // SALARIED_H
```

شکل ۱۵-۱۳ | فایل سرآیند کلاس SalariedEmployee.

```
1  // Fig. 13.16: SalariedEmployee.cpp
2  // SalariedEmployee class member-function definitions.
3  #include <iostream>
4  using std::cout;
5
6  #include "SalariedEmployee.h" // SalariedEmployee class definition
7
8  // constructor
9  SalariedEmployee::SalariedEmployee( const string &first,
10    const string &last, const string &ssn, double salary )
11    : Employee( first, last, ssn )
12 {
13    setWeeklySalary( salary );
14 } // end SalariedEmployee constructor
15
16 // set salary
17 void SalariedEmployee::setWeeklySalary( double salary )
18 {
19    weeklySalary = ( salary < 0.0 ) ? 0.0 : salary;
20 } // end function setWeeklySalary
21
22 // return salary
23 double SalariedEmployee::getWeeklySalary() const
24 {
25    return weeklySalary;
26 } // end function getWeeklySalary
27
28 // calculate earnings;
29 // override pure virtual function earnings in Employee
30 double SalariedEmployee::earnings() const
31 {
32    return getWeeklySalary();
```

```
33 } // end function earnings
34
35 // print SalariedEmployee's information
36 void SalariedEmployee::print() const
37 {
38    cout << "salaried employee: ";
39    Employee::print(); // reuse abstract base-class print function
40    cout << "\nweekly salary: " << getWeeklySalary();
41 } // end function print
```

شکل ۱۶-۱۳ | فایل پیاده‌سازی کلاس SalariedEmployee.

به فایل سرآیند در کلاس **SalariedEmployee** توجه کنید که در آن توابع عضو **earnings** و **print** را بصورت **virtual** اعلان کرده‌ایم (خطوط 19-18 از شکل ۱۵-۱۳)، در واقع قرار دادن کلمه کلیدی **virtual** قبل از این توابع عضو اضافی است. ما آنها را بعنوان **virtual** در کلاس مبنای **Employee** اعلان کرده‌ایم، از اینرو آنها در کل سلسله مراتب کلاس بصورت توابع **virtual** باقی خواهد ماند.

تابع **print** از کلاس **SalariedEmployee** (خطوط 41-36 از شکل ۱۶-۱۳) مبادرت به متوقف ساختن تابع **print** از کلاس مبنای **Employee** می‌کند. اگر کلاس **SalaredEmployee** این تابع **print** را متوقف یا **override** نمی‌کرد، این کلاس، نسخه **print** از کلاس **Employee** را به ارث می‌برد. در چنین وضعی، تابع **print** کلاس **SalariedEmployee** فقط نام کامل و شماره تامین اجتماعی کارمند را برگشت می‌داد که نشاندهنده اطلاعات کافی در مورد این نوع کارمند نیست. برای چاپ اطلاعات کامل کارمندی از نوع **SalariedEmployee**، تابع **print** کلاس مشتق شده، عبارت "Salaried employee:" را چاپ و بدنبال آن اطلاعات خاص کلاس مبنای **Employee** (یعنی نام، نام خانوادگی و شماره تامین اجتماعی) را با فراخوانی تابع **print** کلاس مبنا با استفاده از عملگر تفکیک قلمرو (خط **39**) قرار می‌دهد. خروجی تولید شده توسط تابع **print** کلاس **SalariedEmployee** حاوی حقوق هفتگی کارمند است که با فراخوانی تابع **getWeeklySalary** تهیه شده است.

۳-۶-۱۳ ایجاد کلاس مشتق شده غیرانتزاعی HourlyEmployee

کلاس **HourlyEmployee** (شکل‌های ۱۷-۱۳ و ۱۸-۱۳) نیز از کلاس **Employee** مشتق شده است (خط 8 از شکل ۱۷-۱۳) توابع عضو سراسری شامل یک سازنده (خطوط 12-11) هستند که آرگومان‌های بعنوان نام، نام خانوادگی، شماره تامین اجتماعی، دستمزد ساعتی و تعداد ساعات کارکرد در هفته را دریافت می‌کند، توابع *set* که مقادیر جدید را به اعضای داده **wage** و **hours** تخصیص می‌دهند (خطوط 17 و 14)، توابع *get* که مبادرت به برگشت دادن مقادیر **wage** و **hours** می‌کنند (خطوط 15 و 18)، تابع **earnings** که **virtual** بوده و حقوق یک کارمند از نوع **HourlyEmployee** را محاسبه می‌کند (خط 21) و تابع **print** که آن هم **virtual** است و جمله: "hourly employee:"و اطلاعات خاص کارمند را چاپ می‌کند (خط 22).

```
1  // Fig. 13.17: HourlyEmployee.h
2  // HourlyEmployee class definition.
```

```
3  #ifndef HOURLY_H
4  #define HOURLY_H
5
6  #include "Employee.h" // Employee class definition
7
8  class HourlyEmployee : public Employee
9  {
10 public:
11    HourlyEmployee( const string &, const string &,
12       const string &, double = 0.0, double = 0.0 );
13
14    void setWage( double ); // set hourly wage
15    double getWage() const; // return hourly wage
16
17    void setHours( double ); // set hours worked
18    double getHours() const; // return hours worked
19
20    // keyword virtual signals intent to override
21    virtual double earnings() const; // calculate earnings
22    virtual void print() const; // print HourlyEmployee object
23 private:
24    double wage; // wage per hour
25    double hours; // hours worked for week
26 }; // end class HourlyEmployee
27
28 #endif // HOURLY_H
```

شکل ۱۷-۱۳ | فایل سرآیند کلاس HourlyEmployee.

```
1  // Fig. 13.18: HourlyEmployee.cpp
2  // HourlyEmployee class member-function definitions.
3  #include <iostream>
4  using std::cout;
5
6  #include "HourlyEmployee.h" // HourlyEmployee class definition
7
8  // constructor
9  HourlyEmployee::HourlyEmployee( const string &first, const string &last,
10    const string &ssn, double hourlyWage, double hoursWorked )
11    : Employee( first, last, ssn )
12 {
13    setWage( hourlyWage ); // validate hourly wage
14    setHours( hoursWorked ); // validate hours worked
15 } // end HourlyEmployee constructor
16
17 // set wage
18 void HourlyEmployee::setWage( double hourlyWage )
19 {
20    wage = ( hourlyWage < 0.0 ? 0.0 : hourlyWage );
21 } // end function setWage
22
23 // return wage
24 double HourlyEmployee::getWage() const
25 {
26    return wage;
27 } // end function getWage
28
29 // set hours worked
30 void HourlyEmployee::setHours( double hoursWorked )
31 {
32    hours = ( ( ( hoursWorked >= 0.0 ) && ( hoursWorked <= 168.0 ) ) ?
33       hoursWorked : 0.0 );
34 } // end function setHours
35
36 // return hours worked
37 double HourlyEmployee::getHours() const
38 {
39    return hours;
40 } // end function getHours
41
42 // calculate earnings;
```

```
43 // override pure virtual function earnings in Employee
44 double HourlyEmployee::earnings() const
45 {
46    if ( getHours() <= 40 ) // no overtime
47       return getWage() * getHours();
48    else
49       return 40 * getWage() + ( (getHours() - 40) * getWage() * 1.5 );
50 } // end function earnings
51
52 // print HourlyEmployee's information
53 void HourlyEmployee::print() const
54 {
55    cout << "hourly employee: ";
56    Employee::print(); // code reuse
57    cout << "\nhourly wage: " << getWage() <<
58       "; hours worked: " << getHours();
59 } // end function print
```

شکل ۱۸-۱۳ | فایل پیاده‌سازی کلاس HourlyEmployee.

برنامه شکل ۱۸-۱۳ حاوی پیاده‌سازی تابع عضو برای کلاس **HourlyEmployee** است. خطوط 21-18 و 34-30 تعریف کننده توابع *set* هستند که مقادیر جدید را به اعضای داده **wage** و **hours** تخصیص می‌دهند. تابع **setWage** در خطوط 21-18 ما را مطمئن می‌سازد که **wage** یک مقدار غیرمنفی است و تابع **setHours** در خطوط 34-30 هم ما را مطمئن می‌کند که عضو داده **hours** مابین 0 و 168 (مجموع کل ساعات کارکرد در یک هفته) قرار دارد. توابع *get* در خطوط 27-24 و 40-37 پیاده‌سازی شده‌اند. این توابع را بصورت **virtual** اعلان نکرده‌ایم، از اینرو کلاس‌های مشتق شده از کلاس **HourlyEmployee** نمی‌توانند آنها را override کنند. به سازنده **HourlyEmployee** توجه کنید، همانند سازنده **SalariedEmployee**، مبادرت به ارسال نام، نام خانوادگی و شماره تامین اجتماعی به سازنده کلاس مبنا **Employee** می‌کند (خط 11) تا اعضای داده **private** ارث برده شده و اعلان شده در کلاس مبنا مقداردهی اولیه گردد. علاوه بر این، تابع **print** این کلاس مبادرت به فراخوانی تابع **print** کلاس مبنا (خط 56) می‌کند تا اطلاعات خاص کارمند (یعنی نام و نام خانوادگی و شماره تامین اجتماعی) چاپ شود.

۱۳-۶-٤ ایجاد کلاس مشتق شده غیرانتزاعی CommissionEmployee

کلاس **CommissionEmployee** (شکل‌های ۱۹-۱۳ و ۲۰-۱۳) از کلاس **Employee** (خط 8 از شکل ۱۹-۱۳) مشتق شده است. پیاده‌سازی تابع عضو (شکل ۲۰-۱۳) شامل یک سازنده در خطوط 15-9 است که نام، نام خانوادگی، شماره تامین اجتماعی، میزان حقوق و نرخ کمیسیون را اخذ می‌کند، توابع *set* (خطوط 21-18 و 33-30) برای تخصیص مقادیر جدید به اعضای **CommissionRate** و **grossSales** بکار گرفته شده‌اند، توابع *get* (خطوط 27-24 و 39-36) که مقادیر این اعضای داده را بازیابی می‌کنند، تابع **earnings** (خطوط 46-43) برای محاسبه حقوق کارمندی از نوع **CommissionEmployee**، و تابع **print** در خطوط 55-49 که نوع کارمند را بصورت "commission employee:" و اطلاعات خاص

کارمند را چاپ می‌کند. همچنین سازنده **CommissionEmployee** نام، نام خانوادگی و شماره تامین اجتماعی را به سازنده **Employee** در خط 11 ارسال می‌کند تا با اعضاء داده **private** کلاس **Employee** مقداردهی اولیه شوند. تابع **print**، تابع **print** کلاس مبنا را فراخوانی می‌کند (خط 52) تا اطلاعات خاص **Employee** به نمایش در آید (نام، نام خانوادگی و شماره تامین اجتماعی).

```
1  // Fig. 13.19: CommissionEmployee.h
2  // CommissionEmployee class derived from Employee.
3  #ifndef COMMISSION_H
4  #define COMMISSION_H
5
6  #include "Employee.h" // Employee class definition
7
8  class CommissionEmployee : public Employee
9  {
10 public:
11    CommissionEmployee( const string &, const string &,
12       const string &, double = 0.0, double = 0.0 );
13
14    void setCommissionRate( double ); // set commission rate
15    double getCommissionRate() const; // return commission rate
16
17    void setGrossSales( double ); // set gross sales amount
18    double getGrossSales() const; // return gross sales amount
19
20    // keyword virtual signals intent to override
21    virtual double earnings() const; // calculate earnings
22    virtual void print() const; // print CommissionEmployee object
23 private:
24    double grossSales; // gross weekly sales
25    double commissionRate; // commission percentage
26 }; // end class CommissionEmployee
27
28 #endif // COMMISSION_H
```

شکل ۱۹-۱۳ |فایل سرآیند کلاس CommissionEmploye.

```
1  // Fig. 13.20: CommissionEmployee.cpp
2  // CommissionEmployee class member-function definitions.
3  #include <iostream>
4  using std::cout;
5
6  #include "CommissionEmployee.h" // CommissionEmployee class definition
7
8  // constructor
9  CommissionEmployee::CommissionEmployee( const string &first,
10    const string &last, const string &ssn, double sales, double rate )
11    : Employee( first, last, ssn )
12 {
13    setGrossSales( sales );
14    setCommissionRate( rate );
15 } // end CommissionEmployee constructor
16
17 // set commission rate
18 void CommissionEmployee::setCommissionRate( double rate )
19 {
20     commissionRate = ( ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0 );
21 } // end function setCommissionRate
22
23 // return commission rate
24 double CommissionEmployee::getCommissionRate() const
25 {
26     return commissionRate;
27 } // end function getCommissionRate
28
29 // set gross sales amount
30 void CommissionEmployee::setGrossSales( double sales )
```

```
31 {
32    grossSales = ( ( sales < 0.0 ) ? 0.0 : sales );
33 } // end function setGrossSales
34
35 // return gross sales amount
36 double CommissionEmployee::getGrossSales() const
37 {
38     return grossSales;
39 } // end function getGrossSales
40
41 // calculate earnings;
42 // override pure virtual function earnings in Employee
43 double CommissionEmployee::earnings() const
44 {
45    return getCommissionRate() * getGrossSales();
46 } // end function earnings
47
48 // print CommissionEmployee's information
49 void CommissionEmployee::print() const
50 {
51    cout << "commission employee: ";
52    Employee::print(); // code reuse
53    cout << "\ngross sales: " << getGrossSales()
54       << "; commission rate: " << getCommissionRate();
55 } // end function print
```

شکل ۲۰-۱۳ | فایل پیاده‌سازی کلاس CommssionEmployee.

۵-۶-۱۳ ایجاد غیرمستقیم کلاس مشتق شده غیرانتزاعی BasePlusCommissionEmployee

کلاس **BasePlusCommissionEmployee** (شکل‌های ۲۱-۱۳ و ۲۲-۱۳) بطور مستقیم از کلاس **CommissionEmployee** (خط 8 از شکل ۲۱-۱۳) ارث‌بری دارد و بنابر این یک کلاس مشتق شده غیرمستقیم از کلاس **Employee** است. پیاده‌سازی تابع عضو کلاس **BasePlusCommissionEmployee** شامل یک سازنده (خط 16-10 از شکل ۲۲-۱۳) است که آرگومان‌های بعنوان نام، نام خانوادگی. شماره تامین اجتماعی، میزان حقوق، نرخ کمیسیون و حقوق پایه دریافت می‌کند. سپس نام، نام خانوادگی، شماره تامین اجتماعی، میزان حقوق و نرخ کمیسیون را به سازنده **CommssionEmployee** ارسال می‌کند (خط 13) تا اعضای به ارث رفته مقداردهی اولیه شوند. همچنین کلاس **BasePlusCommissionEmployee** حاوی یک تابع *set* (خطوط 22-19) برای تخصیص مقدار جدید به عضو داده **baseSalary** و یک تابع *get* (خطوط 28-25) برای برگشت دادن مقدار **baseSalary** است. تابع **earnings** در خطوط 35-32 حقوق کارمندی از این نوع را محاسبه می‌کند. توجه کنید که خط 34 در تابع **earnings** تابع **earnings** کلاس مبنای **CommissionEmployee** را برای محاسبه آن بخش از حقوق را که از کمیسیون تامین می‌شود، فراخوانی می‌کند. تابع **print** کلاس **BasePlusCommissionEmployee** (خطوط 43-38) جمله "base-salaried" و بدنبال آن خروجی تابع **print** کلاس مبنای **CommissionEmployee** را چاپ می‌کند. در نتیجه خروجی شامل جمله: "base-salaried commission employee:" و بدنبال آن مابقی اطلاعات **BasePlusCommissionEmployee** خواهد بود.

```
1  // Fig. 13.21: BasePlusCommissionEmployee.h
```

```
2  // BasePlusCommissionEmployee class derived from Employee.
3  #ifndef BASEPLUS_H
4  #define BASEPLUS_H
5
6  #include "CommissionEmployee.h" // CommissionEmployee class definition
7
8  class BasePlusCommissionEmployee : public CommissionEmployee
9  {
10 public:
11    BasePlusCommissionEmployee( const string &, const string &,
12       const string &, double = 0.0, double = 0.0, double = 0.0 );
13
14    void setBaseSalary( double ); // set base salary
15    double getBaseSalary() const; // return base salary
16
17    // keyword virtual signals intent to override
18    virtual double earnings() const; // calculate earnings
19    virtual void print() const; //print BasePlusCommissionEmployee object
20 private:
21    double baseSalary; // base salary per week
22 }; // end class BasePlusCommissionEmployee
23
24 #endif // BASEPLUS_H
```

شکل ۲۱-۱۳ | فایل سرآیند کلاس BasePlusCommissionEmployee.

```
1  // Fig. 13.22: BasePlusCommissionEmployee.cpp
2  // BasePlusCommissionEmployee member-function definitions.
3  #include <iostream>
4  using std::cout;
5
6  // BasePlusCommissionEmployee class definition
7  #include "BasePlusCommissionEmployee.h"
8
9  // constructor
10 BasePlusCommissionEmployee::BasePlusCommissionEmployee(
11    const string &first, const string &last, const string &ssn,
12    double sales, double rate, double salary )
13    : CommissionEmployee( first, last, ssn, sales, rate )
14 {
15    setBaseSalary( salary ); // validate and store base salary
16 } // end BasePlusCommissionEmployee constructor
17
18 // set base salary
19 void BasePlusCommissionEmployee::setBaseSalary( double salary )
20 {
21    baseSalary = ( ( salary < 0.0 ) ? 0.0 : salary );
22 } // end function setBaseSalary
23
24 // return base salary
25 double BasePlusCommissionEmployee::getBaseSalary() const
26 {
27     return baseSalary;
28 } // end function getBaseSalary
29
30 // calculate earnings;
31 // override pure virtual function earnings in Employee
32 double BasePlusCommissionEmployee::earnings() const
33 {
34     return getBaseSalary() + CommissionEmployee::earnings();
35 } // end function earnings
36
37 // print BasePlusCommissionEmployee's information
38 void BasePlusCommissionEmployee::print() const
39 {
40    cout << "base-salaried ";
41    CommissionEmployee::print(); // code reuse
42    cout << "; base salary: " << getBaseSalary();
43 } // end function print
```

شکل ۲۲-۱۳ | فایل پیاده‌سازی کلاس BasePlusCommissionEmployee.

بخاطر دارید که تابع **print** کلاس **CommissionEmployee** مبادرت به نمایش نام، نام خانوادگی و شماره تامین اجتماعی کارمند با فراخوانی تابع **print** از کلاس مبنای خود (یعنی **Employee**) می‌کرد. دقت کنید که تابع **print** کلاس **BasePlusCommissionEmployee** باعث راه‌اندازی زنجیره‌ای از فراخوانی‌های توابع می‌شود که در هر سه سطح سلسله مراتب **Employee** گسترش می‌یابد.

۶-۶-۱۳ شرح فرآیند چند ریختی

برای تست سلسله مراتب **Employee**، برنامه موجود در شکل ۲۳-۱۳ مبادرت به ایجاد یک شی از هر چهار شکل غیرانتزاعی بنام‌های **SalariedEmployee**، **HourlyEmployee**، **CommissionEmployee** و **BasePlusCommissionEmployee** می‌کند. برنامه با این شی‌ها کار می‌کند، ابتدا به روش مقیدسازی استاتیک، سپس چندریختی، با استفاده از برداری از اشاره‌گرهای **Employee**. خطوط 38-31 شی‌های از چهار کلاس غیرانتزاعی مشتق شده از کلاس **Employee** ایجاد می‌کنند. خطوط 51-43 اطلاعات و حقوق هر کارمند را در خروجی به نمایش در می‌آورند.

```
1  // Fig. 13.23: fig13_23.cpp
2  // Processing Employee derived-class objects individually
3  // and polymorphically using dynamic binding.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7  using std::fixed;
8
9  #include <iomanip>
10 using std::setprecision;
11
12 #include <vector>
13 using std::vector;
14
15 // include definitions of classes in Employee hierarchy
16 #include "Employee.h"
17 #include "SalariedEmployee.h"
18 #include "HourlyEmployee.h"
19 #include "CommissionEmployee.h"
20 #include "BasePlusCommissionEmployee.h"
21
22 void virtualViaPointer( const Employee * const ); // prototype
23 void virtualViaReference( const Employee & ); // prototype
24
25 int main()
26 {
27    // set floating-point output formatting
28    cout << fixed << setprecision( 2 );
29
30    // create derived-class objects
31    SalariedEmployee salariedEmployee(
32       "John", "Smith", "111-11-1111", 800 );
33    HourlyEmployee hourlyEmployee(
34       "Karen", "Price", "222-22-2222", 16.75, 40 );
35    CommissionEmployee commissionEmployee(
36       "Sue", "Jones", "333-33-3333", 10000, .06 );
37    BasePlusCommissionEmployee basePlusCommissionEmployee(
38       "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );
39
40    cout << "Employees processed individually using static binding:\n\n";
41
42    // output each Employee's information and earnings using static binding
43    salariedEmployee.print();
```

```
44    cout << "\nearned $" << salariedEmployee.earnings() << "\n\n";
45    hourlyEmployee.print();
46    cout << "\nearned $" << hourlyEmployee.earnings() << "\n\n";
47    commissionEmployee.print();
48    cout << "\nearned $" << commissionEmployee.earnings() << "\n\n";
49    basePlusCommissionEmployee.print();
50    cout << "\nearned $" << basePlusCommissionEmployee.earnings()
51       << "\n\n";
52
53    // create vector of four base-class pointers
54    vector < Employee * > employees( 4 );
55
56    // initialize vector with Employees
57    employees[ 0 ] = &salariedEmployee;
58    employees[ 1 ] = &hourlyEmployee;
59    employees[ 2 ] = &commissionEmployee;
60    employees[ 3 ] = &basePlusCommissionEmployee;
61
62    cout<< "Employees processed polymorphically via dynamic binding:\n\n";
63
64    // call virtualViaPointer to print each Employee's information
65    // and earnings using dynamic binding
66    cout << "Virtual function calls made off base-class pointers:\n\n";
67
68    for ( size_t i = 0; i < employees.size(); i++ )
69       virtualViaPointer( employees[ i ] );
70
71    // call virtualViaReference to print each Employee's information
72    // and earnings using dynamic binding
73    cout << "Virtual function calls made off base-class references:\n\n";
74
75    for ( size_t i = 0; i < employees.size(); i++ )
76       virtualViaReference( *employees[ i ] ); // note dereferencing
77
78    return 0;
79 } // end main
80
81 // call Employee virtual functions print and earnings off a
82 // base-class pointer using dynamic binding
83 void virtualViaPointer( const Employee * const baseClassPtr )
84 {
85    baseClassPtr->print();
86    cout << "\nearned $" << baseClassPtr->earnings() << "\n\n";
87 } // end function virtualViaPointer
88
89 // call Employee virtual functions print and earnings off a
90 // base-class reference using dynamic binding
91 void virtualViaReference( const Employee &baseClassRef )
92 {
93    baseClassRef.print();
94    cout << "\nearned $" << baseClassRef.earnings() << "\n\n";
95 } // end function virtualViaReference
```

```
Employee processed individually using ststic binding:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: 16.75; hours worked: 40.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary:300.00
```

```
earned $500.00

Employee processed polymorphically using dynamic binding:

Virtual function calls made off base-class pointers:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: 16.75; hours worked: 40.00
earned $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary:300.00
earned $500.00

Virtual function calls made off base-class references:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: 16.75; hours worked: 40.00
earned $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary:300.00
earned $500.00
```

شکل ۲۳-۱۳ | برنامه راه‌انداز سلسله مراتب کلاس Employee.

هر تابع عضو اح‌ض‌ار شده در خطوط 51-43 مثالی از مقیدسازی استاتیک در زمان کامپایل است، چرا که از اسامی دستگیرها (و نه اشاره‌گرها یا مراجعه‌ها) استفاده کرده‌ایم. کامپایلر قادر به شناسایی نوع هر شی بود و تعیین می‌کند که کدام تابع **print** و **earnings** فراخوانی شده است.

در خط 54 بردار **employee** اخذ شده که حاوی چهار اشاره‌گر **Employees** است. خط 57 مبادرت به هدایت **employees[0]** بطرف شی **SalariedEmployee** می‌کند. خط 58 مبادرت به هدایت **employees[1]** بطرف شی **hourlyEmployee**، خط 59 مبادرت به هدایت **employees[2]** بطرف شی **CommissionEmployee** و خط 60 مبادرت به هدایت **employees[3]** بطرف شی **BasePlusCommissionEmployee** کرده است. کامپایلر اجازه چنین تخصیص‌هایی را می‌دهد، چرا که

SalariedEmployee یک **Employee** است، **HourlyEmployee** یک **Employee** است، یک **CommissionEmployee** یک **Employee** است و یک **BasePlusCommissionEmployee** نیز یک **Employee** می‌باشد. بنابر این، می‌توانیم آدرس‌های **SalariedEmployee**، **HourlyEmployee**، **CommissionEmployee** و **BasePlusCommissionEmployee** را به اشاره‌گرهای کلاس مبنا **Employee** تخصیص دهیم.

عبارت **for** در خطوط 69-68 بردار **employees** را پیمایش کرده و تابع **virtualViaPointer** (خطوط 87-83) را برای هر عنصر موجود در **employees** فراخوانی می‌کند. تابع **virtualViaPointer** در پارامتر **baseClassPtr** (از نوع **const Employee * const**) آدرس ذخیره شده در یک عنصر **employees** را دریافت می‌کند. در هر بار فراخوانی، این تابع از **baseClassPtr** برای فراخوانی توابع **print** و **earnings** که مجازی یا **virtual** هستند استفاده می‌کند (خطوط 85 و 86). توجه کنید که تابع **virtualViaPointer** حاوی هیچ داده‌ای در ارتباط با نوع **SalariedEmployee**، **HourlyEmployee**، **CommissionEmployee** یا **BasePlusCommissionEmployee** نیست. تابع فقط اطلاعاتی در مورد نوع کلاس مبنا یعنی **Employee** دارد. بنابر این، در زمان کامپایل، کامپایلر نمی‌داند که کدام توابع کلاس غیرانتزاعی را از طریق **baseClassPtr** فراخوانی نماید. با اینحال در زمان اجرا، هر احضار تابع مجازی مبادرت به فراخوانی تابع بر روی شی می‌کند که **baseClassPtr** در آن زمان به آن اشاره دارد. خروجی برنامه نشان می‌دهد که براستی توابع مقتضی برای هر کلاس احضار شده و اطلاعات صحیح هر شی بنمایش درآمده است. برای نمونه، حقوق هفتگی برای کلاس **SalaryEmployee** بنمایش در آمده و ناخالص فروش برای **CommissionEmployee** و **BasePlusCommissionEmployee** نشان داده شده است. همچنین به محاسبه حقوق هر کارمند به روش چند ریختی در خط 86 توجه کنید که همان نتایج تولیدی به روش مقیدسازی استاتیکی است که در خطوط 44، 46 و 50 بدست آمده است.

در پایان، از یک عبارت **for** دیگر (خطوط 76-75) برای پیمایش **employees** و احضار تابع **virtualViaReference** بر روی هر عنصر در بردار استفاده شده است (خطوط 95-91). تابع **virtualViaReference** در پارامتر **baseClassRef** خود (از نوع **const Employee &**) یک مراجعه بفرم *dereference* کردن اشاره‌گر (یعنی دستیابی به اطلاعات از طریق آدرس موجود در یک اشاره‌گر) ذخیره شده در هر عنصر **employees** می‌پردازد (خط 76). در هر بار فراخوانی **virtualViaReference** توابع **print** و **earnings** که **virtual** هستند (خطوط 94-93) از طریق مراجعه **baseClassRef** احضار می‌شوند تا ثابت شود که پردازش چند ریختی با مراجعه‌های کلاس مبنا نیز بخوبی اتفاق می‌افتد. با احضار هر تابع **virtual**، تابعی بر روی شی که **baseClassRef** در زمان اجرا به آن اشاره دارد، فراخوانی می‌شود.

این مثالی دیگر از مقیدسازی دینامیکی است. خروجی تولید شده توسط مراجعه‌های کلاس مبنا با خروجی تولید شده توسط اشاره‌گرهای کلاس مبنا یکسان است.

۱۳-۷ چند ریختی، توابع virtual و مقیدسازی دینامیکی

C++ بکارگیری روش چندریختی در برنامه‌ها را آسانتر کرده است. بطور کاملاً مشخص امکان استفاده از روش چندریختی در زبان‌های غیر شی‌گرا همانند C وجود دارد، اما انجام اینکار مستلزم پیچیدگی کار با اشاره‌گرها است که خود خطرات بالقوه‌ای برای برنامه می‌تواند داشته باشد.

در این بخش به بررسی نحوه عملکرد داخلی C++ در پیاده‌سازی چند ریختی، توابع **virtual** و مقیدسازی دینامیکی می‌پردازیم. با مطالعه این بخش اطلاعات اولیه و خوبی از نحوه کار این قابلیت‌ها بدست خواهید آورد. از همه مهمتر، این بخش به شما کمک می‌کند از کاری که چندریختی برایتان (منظور هزینه چندریختی است) در مصرف حافظه و زمان پردازنده انجام می‌دهد. مطلع شوید. همچنین به شما کمک می‌کند تا مشخص نمائید در چه مواقعی از چند ریختی استفاده کنید و در چه مواقعی آنرا به کنار بگذارید. ابتدا به توضیح ساختمان داده‌ای می‌پردازیم که کامپایلر C++ در زمان اجرا به منظور پشتیبانی از چندریختی در زمان اجرا، تهیه می‌کند. مشاهده خواهید کرد که چند ریختی از طریق سه سطح اشاره‌گر ("سه‌گانه غیرمستقیم") صورت می‌گیرد. سپس نشان خواهیم داد که چگونه در زمان اجرا از این ساختارهای داده برای اجرای توابع **virtual** و انجام مقیدسازی دینامیکی مرتبط با چند ریختی استفاده می‌کند. توجه کنید که بحث ما یکی از روش‌های ممکنه پیاده‌سازی است و نه جزء اصول پیاده‌سازی زبان. زمانیکه C++ مبادرت به کامپایل کلاسی می‌کند که دارای یک یا چندین تابع **virtual** است، یک *جدول تابع مجازی* یا *vtable* (virtual function table) برای آن کلاس ایجاد می‌کند. برنامه در هر بار اجرا، از این vtable برای انتخاب تابع صحیح، هر بار که یک تابع **virtual** برای آن کلاس فراخوانی می‌شود، استفاده می‌کند. سمت چپ‌ترین ستون در شکل ۱۳-۲۴ نشاندهنده vtable برای کلاس‌های **Employee**, **SalariedEmployee**، **HourlyEmployee**، **CommissionEmployee** و **BasePlusCommissionEmployee** است.

در vtable کلاس **Employee**، اشاره‌گر تابع اول با صفر تنظیم شده است (یعنی اشاره‌گر null). اینکار به این دلیل صورت گرفته که تابع **earnings** یک تابع **virtual** محض است و بنابر این فاقد پیاده‌سازی می‌باشد. اشاره‌گر تابع دوم به تابع **print** اشاره دارد، که نام کامل و شماره تامین اجتماعی کارمند را به نمایش در می‌آورد. هر کلاسی که دارای یک یا چند اشاره‌گر **null** در جدول vtable خود است یک کلاس انتزاعی می‌باشد. کلاس‌های که فاقد null در vtable هستند (همانند **SalareiedEmployee**،

HourlyEmployee، **CommissionEmployee** و **BasePlusCommissionEmployee**) کلاس‌های غیرانتزاعی می‌باشند.

کلاس **SalariedEmployee** مبادرت به override کردن تابع **earnings** می‌کند تا حقوق هفتگی کارمند را برگشت دهد، از اینرو اشاره‌گر تابع به تابع **earnings** از کلاس **SalariedEmployee** اشاره می‌کند. همچنین این کلاس مبادرت به override کردن تابع **print** می‌کند تا اشاره‌گر تابع متناظر به تابع عضو **SalariedEmployee** اشاره کند که جمله "salaried employee:" و بدنبال آن نام کارمند، شماره تامین اجتماعی و حقوق هفتگی چاپ شود.

اشاره‌گر تابع **earnings** در vtable کلاس **HourlyEmployee** به تابع **earnings** کلاس **HourlyEmployee** اشاره دارد که حاصلضرب دستمزد (wage) کارمند به تعداد ساعات (hours) کار را برگشت می‌دهد. اشاره‌گر تابع **print** به نسخه **HourlyEmployee** تابع اشاره دارد که جمله: "hourly employee:"، نام کارمند، شماره تامین اجتماعی، دستمزد ساعتی و ساعات کارکرد را چاپ می‌کند. هر دو مبادرت به override کردن توابع در کلاس **Employee** می‌کنند.

اشاره‌گر تابع **earnings** در vtable برای کلاس **CommissionEmployee** به تابع **earnings** این کلاس اشاره می‌کند که حاصلضرب فروش ناخالص در نرخ کمیسیون را برگشت می‌دهد. اشاره‌گر تابع **print** به نسخه **CommissionEmployee** تابع اشاره دارد که نوع کارمند، نام، شماره تامین اجتماعی، نرخ کمیسیون و فروش ناخالص را چاپ می‌کند. همانند کلاس **HourlyEmployee** هر دو تابع مبادرت به override کردن توابع در کلاس **Employee** می‌کنند.

اشاره‌گر تابع **earnings** در vtable برای کلاس **BasePlusCommissionEmployee** به تابع **earnings** این کلاس اشاره می‌کند که حاصل حقوق پایه به همراه فروش ناخالص ضرب شده در نرخ کمیسیون را برگشت می‌دهد. اشاره‌گر تابع **print** به نسخه **BasePlusCommissionEmployee** تابع اشاره دارد که نوع این کارمند، نام، شماره تامین اجتماعی، نرخ کمیسیون و فروش ناخالص را چاپ می‌کند. هر دو تابع مبادرت به override توابع در کلاس **CommissionEmployee** می‌کنند.

شکل ۲۴-۱۳ | نحوه عملکرد فراخوانی تابع virtual.

اگر به مبحث آموزشی **Empolyee** توجه کنید متوجه می‌شوید که هر کلاس غیرانتزاعی دارای پیاده‌سازی متعلق بخود برای توابع **virtual** بنام‌های **print** و **earnings** است. تا بدین جا آموخته‌اید که هر کلاسی که مستقیماً از کلاس مبنای انتزاعی **Employee** ارث‌بری دارد، بایستی تابع **earnings** را به نحوی پیاده کند که یک کلاس غیرانتزاعی گردد، چرا که **earnings** یک تابع **virtual** محض می‌باشد. این کلاس‌ها نیازی ندارند تا تابع **print** را پیاده‌سازی کنند، با این همه، با توجه به غیرانتزاعی بودن آنها، تابع یک تابع

virtual محض نیست و کلاس‌های مشتق شده می‌توانند پیاده‌سازی **print** در کلاس **Employee** را به ارث ببرند. از این گذشته، **BasePlusCommissionEmployee** مجبور نیست تا تابع **print** یا **earnings** را پیاده‌سازی کند، پیاده‌سازی هر دو تابع را می‌تواند از کلاس **CommissionEmployee** به ارث ببرد. اگر کلاسی در سلسله مراتب ما از پیاده‌سازی توابع به این روش ارث‌بری داشته باشد، اشاره‌گرهای vtable برای این توابع می‌توانند بسادگی به پیاده‌سازی تابعی اشاره داشته باشند که به ارث برده شده است. برای مثال، اگر **BasePlusCommissionEmployee** مبادرت به override کردن تابع **earnings** نکند، اشاره‌گر تابع **earnings** در vtable کلاس **BasePlusCommissionEmployee** به همان تابع **earnings** که در vtable کلاس **CommissionEmployee** به آن اشاره می‌کند، اشاره خواهد کرد.

چند ریختی از طریق یک ساختمان داده کارا و با سه سطح از اشاره‌گر انجام می‌شود. در ارتباط با یک سطح صحبت می‌کنیم، اشاره‌گرهای تابع در vtable. اینها به توابع واقعی اشاره دارند که به هنگام احضار یک تابع **virtual** اجرا می‌شوند.

حال به دومین سطح از اشاره‌گرها می‌پردازیم. هنگامی که یک شی از یک کلاس با یک یا چند تابع **virtual** معرفی می‌شود، کامپایلر مبادرت به الصاق یک اشاره‌گر به شی از جدول vtable برای آن کلاس می‌کند. معمولاً این اشاره‌گر جلوتر از شی قرار می‌گیرد، اما برای پیاده‌سازی به این روش ضرورتی ندارد. در شکل ۲۴-۱۳ این اشاره‌گرها مرتبط با شی‌های ایجاد شده در شکل ۲۳-۱۳ هستند (یک شی برای هر نوع **SalariedEmployee**، **HourlyEmployee**، **CommissionEmployee** و **BasePlusCommissionEmployee**).

دقت کنید که دیاگرام، مقادیر هر عضو داده شی را به نمایش در آورده است. برای مثال، شی **SalariedEmployee** حاوی یک اشاره‌گر به vtable این کلاس بوده، همچنین این شی حاوی مقادیر John Smith، 111-11-1111 و $800.00 است.

سطح سوم اشاره‌گرها فقط حاوی دستگیره‌ها به شی‌های است که فراخوانی تابع **virtual** را دریافت می‌کنند. دستگیره‌ها در این سطح می‌توانند مراجعه باشند. دقت کنید که در شکل ۲۴-۱۳ بردار **employess** رسم شده حاوی اشاره‌گرهای **Employee** می‌باشد. حال اجازه دهید تا به بررسی نحوه اجرای یک تابع **virtual** بپردازیم. به فراخوانی **baseClassPtr->print()** در تابع **virtualViaPointer** در خط 85 از شکل ۲۳-۱۳ توجه کنید. فرض کنید که **baseClassPtr** حاوی **employees[1]** است (یعنی آدرس شی **hourlyEmployee** در **employees**). زمانیکه کامپایلر این عبارت را کامپایل می‌کند، تعیین می‌کند که براستی فراخوانی صورت گرفته از طریق اشاره‌گر کلاس مبنا بوده و اینکه **print** یک تابع **virtual** است.

کامپایلر تعیین می‌کند که **print**، دومین ورودی یا چیز ثبت شده در vtable است. برای تعیین محل این ورودی، کامپایلر متوجه می‌شود که نیاز دارد تا از ورودی اول پرش کند. بنابر این، کامپایلر مبادرت به کامپایل یک افست یا جابجایی به میزان چهار بایت (چهار بایت برای هر اشاره‌گر بر روی اکثر کامپیوترهای 32 بیتی) در جدول اشاره‌گرهای کد شی زبان ماشین می‌کند تا کدی که فراخوانی تابع **virtual** را اجرا خواهد کرد بدست آید.

کد تولیدی توسط کامپایلر عملیات‌های زیر را انجام می‌دهد [نکته: اعداد بکار رفته در لیست متناظر با اعداد موجود در دوایر شکل ۲۴-۱۳ هستند].

۱- انتخاب *i*th ورودی از **employees** (در این مورد آدرس شی **hourlyEmployee**)، و ارسال آن بعنوان یک آرگومان به تابع **virtualViaPointer**. این کار سبب تنظیم پارامتر **baseClassPtr** برای اشاره به **hourlyEmployee** می‌شود.

۲- دستیابی به اطلاعات از طریق آدرس موجود در اشاره‌گر برای بدست آوردن شی **hourlyEmployee**.

۳- دستیابی به اطلاعات اشاره‌گر **hourlyEmployee** در جدول vtable برای رسیدن به vtable **HourlyEmployee**.

۴- جابجایی به میزان چهار بایت برای انتخاب اشاره‌گر تابع **print**.

۵- دستیابی به اطلاعات آدرس اشاره‌گر تابع **print** بفرم نام تابع اصلی که اجرا خواهد شد و استفاده از عملگر فراخوانی تابع ( ) برای اجرای تابع **print** مقتضی که در این مورد چاپ نوع کارمند، نام، شماره تامین اجتماعی، دستمزد ساعتی و ساعت کارکرد در هفته است. امکان دارد ساختمان داده بکار رفته در شکل ۲۴-۱۳ کمی پیچیده بنظر برسد، اما این پیچیدگی توسط کامپایلر مدیریت شده و از دید شما که سرگرم برنامه‌نویسی چندریختی هستید، پنهان است. عملیات دستیابی به اطلاعات از طریق آدرس موجود در اشاره‌گر و دسترسی به حافظه که در هر فراخوانی تابع **virtual** صورت می‌گیرد، مستلزم صرف زمان اضافی در اجرا است. vtable و اشاره‌گرهای vtable که به شی‌ها افزوده می‌شوند هم نیازمند حافظه هستند. با توجه به این موارد می‌توانید مشخص نمائید که آیا استفاده از توابع **virtual** در برنامه‌ها به نفع شما هست یا خیر.

۱۳-۸ مبحث آموزشی: سیستم پرداخت حقوق با استفاده از چند ریختی و اطلاعات نوع زمان اجرا با تبدیل dynamic_cast, typeid و type_info

به صورت مسئله مطرح شده در ابتدای بخش ۶-۱۳ مجدداً توجه کنید که در آن برای یک دوره پرداخت حقوق، شرکت تصمیم گرفته به حقوق پایه کارمندان نوع **BasePlusCommissionEmployee**، ده درصد اضافه کند. در زمان پردازش شی‌های **Employee** به روش چند ریختی در بخش ۶-۶-۱۳، نگران حالت

خاص نبودیم. با این وجود، هم اکنون برای تعدیل کردن حقوق پایه کارمندان **BasePlusCommissionEmployee**، مجبور هستیم تا نوع خاص هر شی **Employee** را در زمان اجرا مشخص کرده، سپس بر اساس آن عمل کنیم. در این بخش به بررسی قابلیت قدرتمندی بنام اطلاعات نوع زمان اجرا یا RTTI[1] و تبدیل دینامیکی خواهیم پرداخت که به برنامه امکان می‌دهند تا نوع یک شی را در زمان اجرا مشخص کرده و مطابق آن شی عمل شود.

برخی از کامپایلرها همانند Microsoft Visual C++ .NET، مستلزم این هستندکه RTTI قبل از اینکه بتواند در برنامه بکار گرفته شود، فعال گردد. می‌توانید با مراجعه به مستندات کامپایلر خود، با نحوه انجام اینکار آشنا شوید. برای فعال کردن RTTI در Visual C++ .NET، از منوی **Project** گزینه خصوصیات یا **Properties** را برای پروژه جاری انتخاب کنید. در کادر تبادلی **Property Pages**، گزینه **Configuration Properties>C/C++>Language** را انتخاب نمائید سپس از جعبه کامبو که در کنار **Enable Run-Time Type Info** قرار گرفته گزینه **Yes(/GR)** را انتخاب کرده و در پایان بر روی دکمه Ok کلیک کنید تا تغیرات صورت گرفته ذخیره شود.

در برنامه شکل ۲۵-۱۳ از سلسله مراتب **Employee** که در بخش ۶-۱۳ توسعه یافته استفاده کرده و ده درصد به حقوق پایه هر **BasePlusCommissionEmployee** اضافه می‌کنیم.

```
1  // Fig. 13.25: fig13_25.cpp
2  // Demonstrating downcasting and run-time type information.
3  // NOTE: For this example to run in Visual C++ .NET,
4  // you need to enable RTTI (Run-Time Type Info) for the project.
5  #include <iostream>
6  using std::cout;
7  using std::endl;
8  using std::fixed;
9
10 #include <iomanip>
11 using std::setprecision;
12
13 #include <vector>
14 using std::vector;
15
16 #include <typeinfo>
17
18 // include definitions of classes in Employee hierarchy
19 #include "Employee.h"
20 #include "SalariedEmployee.h"
21 #include "HourlyEmployee.h"
22 #include "CommissionEmployee.h"
23 #include "BasePlusCommissionEmployee.h"
24
25 int main()
26 {
27    // set floating-point output formatting
28    cout << fixed << setprecision( 2 );
29
30    // create vector of four base-class pointers
31    vector < Employee * > employees( 4 );
32
```

[1] - Run-Time Type Information

```
33    // initialize vector with various kinds of Employees
34    employees[ 0 ] = new SalariedEmployee(
35       "John", "Smith", "111-11-1111", 800 );
36    employees[ 1 ] = new HourlyEmployee(
37       "Karen", "Price", "222-22-2222", 16.75, 40 );
38    employees[ 2 ] = new CommissionEmployee(
39       "Sue", "Jones", "333-33-3333", 10000, .06 );
40    employees[ 3 ] = new BasePlusCommissionEmployee(
41       "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );
42
43    // polymorphically process each element in vector employees
44    for ( size_t i = 0; i < employees.size(); i++ )
45    {
46       employees[ i ]->print(); // output employee information
47       cout << endl;
48
49       // downcast pointer
50       BasePlusCommissionEmployee *derivedPtr =
51          dynamic_cast < BasePlusCommissionEmployee * >
52             ( employees[ i ] );
53
54       // determine whether element points to base-salaried
55       // commission employee
56       if ( derivedPtr != 0 ) // 0 if not a BasePlusCommissionEmployee
57       {
58          double oldBaseSalary = derivedPtr->getBaseSalary();
59          cout << "old base salary: $" << oldBaseSalary << endl;
60          derivedPtr->setBaseSalary( 1.10 * oldBaseSalary );
61          cout << "new base salary with 10% increase is: $"
62             << derivedPtr->getBaseSalary() << endl;
63       } // end if
64
65       cout << "earned $" << employees[ i ]->earnings() << "\n\n";
66    } // end for
67
68    // release objects pointed to by vector's elements
69    for ( size_t j = 0; j < employees.size(); j++ )
70    {
71       // output class name
72       cout << "deleting object of "
73          << typeid( *employees[ j ] ).name() << endl;
74
75       delete employees[ j ];
76    } // end for
77
78    return 0;
79 } // end main
```

```
salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: 16.75; hours worked: 40.00
earned $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary:300.00
old base salary: $300.00
new base salary with 10% increase is: $330.00
earned $530.00

deleting object of class SalariedEmployee
```

```
deleting object of class HourlyEmployee
deleting object of class CommissionEmployee
deleting object of class BasePlusCommissionEmployee
```

شکل ۲۵-۱۳ | توضیح روش تبدیل و اطلاعات نوع زمان اجرا.

خط 31 چهار عنصر برداری **employees** اعلان کرده که اشاره‌گرهایی به شی‌های **Employee** را ذخیره می‌سازد. خطوط 41-34 بردار را با آدرس‌های اخذ شده دینامیکی شی‌ها از کلاس **SalariedEmployee**(شکل‌های۱۵-۱۳و۱۶-۱۳)، **HourlyEmployee**(شکل‌های۱۷-۱۳و۱۸-۱۳)، **CommissionEmployee** (شکل‌های۱۹-۱۳و ۲۰-۱۳)و **BasePlusCommissionEmployee**(شکل‌های ۲۱-۱۳ و ۲۲-۱۳) پر می‌کند.

عبارت **for** در خطوط 66-44 در میان بردار **employees** حرکت کرده و اطلاعات هر کارمند را با احضار تابع عضو **print** (خط 46) به نمایش در می‌آورد. بخاطر دارید که چون **print** بصورت **virtual** در کلاس مبنا **Employee** اعلان شده، سیستم تابع **print** مناسب با شی کلاس مشتق شده را فراخوانی می‌کند.

در این مثال با شی‌های **BasePlusCommissionEmployee** مواجه شده‌ایم و می‌خواهیم به حقوق پایه آنها ده درصد اضافه کنیم. از آنجائی که پردازش کارمندان را بصورت چند ریختی انجام داده‌ایم، نمی‌توانیم (با تکنیکی که آموخته‌ایم) بطور مشخص نوع **Employee** را برای کار در هر زمان تعیین کنیم. این حالت مشکل‌ساز می‌شود، چرا که کارمندان **BasePlusCommissionEmployee** بایستی در هنگام مواجه شدن با آنها تشخیص داده شوند تا بتوان ده درصد به حقوق آنها اضافه کرد. برای انجام اینکار، از عملگر **dynamic_cast** در خط 51 برای تعیین اینکه آیا نوع شی یک **BasePlusCommissionEmployee** است یا خیر، استفاده کرده‌ایم. در بخش ۳-۳-۱۳ از این نوع تبدیل یاد شده است. خطوط 52-50 بصورت دینامیکی **employees[1]** را از نوع **Employee *** به نوع **BasePlusCommissionEmployee *** تبدیل می‌کند. اگر عنصر **vector** به شیی اشاره کند که یک شی **BasePlusCommissionEmployee** است، پس آدرس آن شی به **commissionPtr** تخصیص داده می‌شود، در غیر اینصورت، صفر به اشاره‌گر **derivedPtr** کلاس مشتق شده تخصیص داده خواهد شد.

اگر مقدار برگشتی توسط عملگر **dynamic_cast** در خطوط 52-50 برابر صفر نباشد، شی از نوع صحیح بوده و عبارت **if** (خطوط 63-56) پردازش خاصی که مورد نیاز شی **BasePlusCommissionEmployee** است انجام می‌دهد. خطوط 58، 60 و 62 توابع **getBaseSalary** و **setBaseSalary** را برای بازیابی و به روز کردن حقوق کارمند فراخوانی می‌کنند.

خط 65 تابع عضو **earnings** را بر روی شی که **employees[1]** به آن اشاره می‌کند، فراخوانی می‌نماید. بخاطر دارید که **earnings** بصورت **virtual** در کلاس مبنا اعلان شده است، از اینرو برنامه تابع **earnings** شی از کلاس مشتق شده را فراخوانی می‌کند. اینحالت نمونه‌ای از مقیدسازی دینامیکی است.

حلقه **for** در خطوط 76-69 نوع هر کارمند را نشان داده و از عملگر **delete** برای آزادسازی یا بازپس‌گیری حافظه دینامیکی که هر عنصر **vector** به آن اشاره می‌کند، استفاده کرده است. عملگر **typeid** در خط 73 یک مراجعه به یک شی از کلاس **type_info** برگشت می‌دهد که حاوی اطلاعاتی در ارتباط با نوع عملوند آن است که شامل نام آن نوع می‌باشد. زمانیکه فراخوانی می‌شود، تابع عضو **type_info** بنام **name** (خط 73) یک رشته مبتنی بر اشاره‌گر برگشت می‌دهد که حاوی نام نوع (مثلاً "*class BasePlusCommissionEmployee*") از آرگومان ارسالی به **typied** است. برای استفاده از **typeid**، برنامه باید دارای فایل سرآیند <**typeinfo**> باشد (خط 6).

توجه کنید با تبدیل اشاره‌گر **Employee** به یک اشاره‌گر **BasePlusCommissionEmployee** (خطوط 52-50) از وقوع چند خطای کامپایل جلوگیری کرده‌ایم. اگر **dynamic_cast** را از خط 51 حذف کرده و مبادرت به تخصیص مستقیم اشاره‌گر جاری **Employee** به اشاره‌گر **commissionPtr** کلاس **BasePlusCommissionEmployee** کنیم، با خطای کامپایلر مواجه خواهیم شد. C++ به برنامه اجازه نمی‌دهد تا اشاره‌گر کلاس مبنا را به اشاره‌گر کلاس مشتق شده تخصیص دهید چرا که در اینحالت رابطه *is-a* نقض می‌شود، **CommissionEmployee** یک **BasePlusCommissionEmployee** نیست. رابطه is-a فقط مابین کلاس مشتق شده و کلاس‌های مبنای آن صادق است و عکس آن برقرار نمی‌باشد.

۹-۱۳ نابود کننده‌های virtual

به هنگام استفاده از روش چند ریختی در پردازش شی‌های اخذ شده دینامیکی از سلسله مراتب، احتمال رخ دادن مشکل وجود دارد. تا بدین جا شاهد نابود کننده‌های غیر virtual بودید، نابود کننده‌های که با کلمه کلیدی virtual اعلان نشده‌اند. اگر یک شی از کلاس مشتق شده با یک نابود کننده غیر virtual بطور صریح نابود شود، با اعمال عملگر delete بر روی اشاره‌گر کلاس مبنا، اینحالت در C++ استاندارد تعریف نشده است.

ساده‌ترین راه حل این مشکل ایجاد یک نابود کننده virtual (یعنی نابود کننده‌ای که با کلمه کلیدی virtual اعلان شده است) در کلاس مبنا است. با اینکار تمام نابود کننده‌های کلاس مشتق شده virtual خواهند شد حتی اگر نام یکسان با نابود کننده کلاس مبنا نداشته باشند. حال اگر یک شی در سلسله مراتب بصورت صریح و با استفاده از عملگر delete بر روی اشاره‌گر کلاس مبنا نابود شود، نابود کننده مقتضی کلاس براساس شی که اشاره‌گر کلاس مبنا به آن اشاره می‌کند، فراخوانی خواهد شد. بخاطر داشته باشید

که به هنگام نابود شدن یک شی از کلاس مشتق شده، بخشی از کلاس مبنا از کلاس مشتق شده نیز از بین می‌رود، از اینرو اجرای هر دو نابود کننده کلاس مشتق شده و مبنا از اهمیت برخوردار است. نابود کننده کلاس مبنا بصورت اتوماتیک پس از اجرای نابود کننده کلاس مشتق شده، اجرا می‌گردد.

خطای برنامه‌نویسی

سازنده‌ها نمی‌توانند **virtual** *باشند. اعلان سازنده بصورت* **virtual** *خطای کامپایل بدنبال خواهد داشت.*

۱۰-۱۳ مبحث آموزشی مهندسی نرم‌افزار: ارثبری در سیستم ATM

در این بخش مجدداً به سراغ طراحی سیستم ATM می‌رویم تا ببینیم چگونه می‌توان از مزایای ارثبری در این سیستم استفاده کرد. برای اعمال توارث ابتدا، نگاهی به نقاط مشترک مابین کلاس‌ها در سیستم می‌اندازیم. یک سلسله مراتب توارث برای مدل کردن کلاس‌ها در فرآیند چندریختی ایجاد می‌کنیم.

سپس دیاگرام کلاس را برای پیوستن روابط ارثبری جدید اصلاح می‌کنیم. در پایان، به بررسی نحوه به روز کردن طراحی خود در تبدیل به فایلهای سرآیند C++ می‌پردازیم.

در بخش ۱۱-۳، با مشکلی مواجه شدیم که در ارتباط با ارائه تراکنش مالی در سیستم بود. بجای ایجاد یک کلاس برای ارائه کلیه تعاملات صورت گرفته، تصمیم گرفتیم تا سه کلاس تراکنشی مجزا بنام‌های **BalanceInquiry**، **Withdrawal** و **Deposit** ایجاد کنیم تا نشاندهنده تعامل‌های باشند که سیستم **ATM** می‌تواند انجام دهد. شکل ۲۶-۱۳ نشاندهنده صفات و عملیات این کلاس‌ها است. توجه کنید که آنها در یک صفت **(accountNumber)** و یک عملیات **(execute)** مشترک هستند. هر کلاسی مستلزم صفت **accountNumber** برای مشخص کردن شماره حساب است تا تراکنش بر آن اعمال شود. هر کلاسی حاوی عملیات **execute** است که **ATM** با فراخوانی آن تراکنش را انجام می‌دهد. واضح است که **BalanceInquiry**، **Withdrawal** و **Deposit** ارائه کننده انواع تراکنش‌ها هستند. شکل ۲۶-۱۳ نقاط مشترک در میان تراکنش کلاس‌ها را آشکار کرده است، از اینرو بنظر می‌رسد استفاده از توارث برای فاکتورگیری از ویژگیهای مشترک، در طراحی این کلاس امکان‌پذیر باشد.

از اینرو عامل مشترک را در کلاس مبنا **Transaction** قرار داده و کلاس **BalanceInquiry**، **Withdrawal** و **Deposit** را از **Transaction** مشتق می کنیم (شکل ۲۷-۱۳).

شکل ۲۶-۱۳ | صفات و عملیات کلاس‌های BalanceInquiry، Withdrawal و Deposit.

UML تصریح کننده یک رابطه بنام *تعمیم* یا *generalization* برای مدل کردن توارث است. شکل ۲۷-۱۳ دیاگرام کلاسی است که رابطه توارث مابین کلاس مبنا **Transactiion** و سه کلاس مشتق شده از آنرا را مدل کرده است. فلش‌های با مثلث‌های توخالی بر این نکته دلالت دارند که کلاس‌های **BalanceInquiry**، **Withdrawal** و **Deposit** از کلاس **Transaction** مشتق شده‌اند. گفته می‌شود

کلاس **Transactiion** به کلاس‌های مشتق شده خودش تعمیم یافته است. گفته می‌شود کلاس‌های مشتق شده ویژه از کلاس **Transaction** هستند.

کلاس‌های **BalanceInquiry**، **Withdrawal** و **Deposit** صفت **accountNumber** از نوع صحیح را به اشتراک می‌گذارند، بنابر این از این صفت مشترک فاکتور گرفته و آنرا در کلاس مبنا **Transaction** جای می‌دهیم. دیگر، در بخش دوم کلاس‌های مشتق شده، **accountNumber** دیده نمی‌شود، چرا که هر سه کلاس مشتق شده این صفت را از **Transaction** ارث می‌برند.

با این همه، بخاطر داشته باشید که کلاس‌های مشتق شده نمی‌توانند به صفات **private** یک کلاس مبنا دسترسی داشته باشند. بنابر این تابع عضو سراسری **getAccountNumber** را در کلاس **Transaction** وارد کرده‌ایم. هر کلاس مشتق شده این تابع عضو را به ارث برده، و می‌تواند به **accountNumber** خود در صورت نیاز به اجرای یک تراکنش دسترسی پیدا کند.

مطابق شکل ۲۶-۱۳، کلاس‌های **BalanceInquiry**، **Withdrawal** و **Deposit** عملیات **execute** را هم به اشتراک گذاشته‌اند، از اینرو کلاس مبنا **Transaction** باید حاوی تابع عضو سراسری **execute** باشد. با این همه، پیاده‌سازی مشترک **execute** در کلاس **Transaction** وجود ندارد چرا که عملکرد این تابع عضو بستگی به نوع خاصی از تراکنش دارد. بنابر این تابع عضو **execute** بعنوان یک تابع **virtual** محض در کلاس مبنا **Transaction** اعلان شده است. با اینکار **Transaction** یک کلاس انتزاعی شده و هر کلاس مشتق شده از **Transaction** بایستی یک کلاس غیرانتزاعی باشد. UML مستلزم این است که اسامی کلاس انتزاعی (و توابع **virtual** محض—عملیات انتزاعی در UML) بصورت ایتالیک نشان داده شوند، از اینرو **Transaction** و تابع عضو آن **execute** بصورت ایتالیک در شکل ۲۷-۱۳ نشان داده شده‌اند. توجه کنید که عملیات **execute** در کلاس‌های مشتق شده **BalanceInquiry**، **Withdrawal** و **Deposit** بصورت ایتالیک نوشته نشده است. هر کلاس مشتق شده مبادرت به override کردن تابع عضو **execurte** کلاس **Transaction** با پیاده‌سازی مقتضی می‌کند. دقت کنید که در شکل ۲۷-۱۳ عملیات **execute** در بخش سوم کلاس‌های مشتق شده جای گرفته است، به این دلیل که هر کلاس دارای یک پیاده‌سازی غیرانتزاعی متفاوت از تابع عضو override شده است.

شکل ۲۷-۱۳ |دیاگرام کلاس مدل کننده رابطه تعمیم مابین کلاس مبنا Transaction و کلاس‌های مشتق شده.

همانطوری که در این فصل آموختید، یک کلاس مشتق شده می‌تواند واسط یا پیاده‌سازی را از کلاس مبنا به ارث ببرد. در مقایسه سلسله مراتب طراحی شده برای پیاده‌سازی توارث، یکی از طرح‌های ارث‌بری واسط متمایل به داشتن عاملیت خود در مراتب پایین سلسله مراتب است. کلاس مبنا دلالت بر یک یا چند

تابع دارد که بایستی توسط هر کلاس در سلسله مراتب تعریف شوند، اما کلاس‌های مشتق شده مجزا از هم پیاده‌سازی متعلق به خود را در ارتباط با توابع تدارک می‌بینند.

طراحی سلسله مراتب توارث صورت گرفته برای سیستم ATM از مزایای این نوع از ارث‌بری استفاده می‌کند و برای ATM روش مناسبی برای اجرای تمام تراکنش‌ها فراهم می‌آورد. هر کلاس مشتق شده از **Transaction** برخی از جزئیات پیاده‌سازی را به ارث می‌برد (مانند عضو داده **accountNumber**)، اما مزیت اصلی همکاری ارث‌بری در سیستم این است که کلاس‌های مشتق شده یک واسط مشترک را به اشتراک می‌گذارند (برای مثال، تابع عضو **execute** که **virtual** است). سیستم ATM می‌تواند اشاره‌گر **Transaction** را بطرف هر تراکنشی هدایت کرده و زمانیکه ATM تابع **execute** را از طریق این اشاره‌گر احضار نماید، نسخه مناسب **execute** برای آن تراکنش بصورت اتوماتیک اجرا می‌شود. برای مثال فرض کنید کاربر گزینه مشاهده میزان موجودی را انتخاب کند ATM اشاره‌گر **Transaction** را بطرف یک شی جدید از کلاس BalanceInquiry هدایت می‌کند، که کامپایلر C++ اجازه آنرا می‌دهد، چرا که **BalanceInquiry** یک شی از **Transaction** می‌باشد (رابطه is-a). زمانیکه ATM از این اشاره‌گر برای احضار **execute** استفاده می‌کند، نسخه **execute** کلاس **BalanceInquiry** فراخوانی می‌گردد.

این روش چند ریختی می‌تواند گسترش و بسط‌پذیری سیستم را به آسانی میسر نماید. برای مثال ممکن است مایل به افزودن یک نوع تراکنش جدید باشیم (برای مثال انتقال وجه یا پرداخت صورتحساب)، کاری که باید انجام دهیم ایجاد یک کلاس مشتق شده از **Transaction** است که مبادرت به override کردن تابع عضو **execute** با نسخه مقتضی برای تراکنش جدید می‌کند. در اینحالت کد سیستم به کمترین تغییر نیاز خواهد داشت.

همانطوری که در ابتدای فصل آموختید، یک کلاس انتزاعی همانند **Transaction** کلاسی است که برنامه‌نویس هرگز قصد نمونه‌سازی از آنرا ندارد. یک کلاس انتزاعی فقط صفات و رفتار مشترک را برای کلاس‌های مشتق شده از خود را در سلسله مراتب توارث اعلان می‌کند. کلاس **Transaction** مفهومی از تراکنش رخ داده بر روی شماره حساب و اجرای تراکنش تعریف می‌کند. ممکن است تعجب کنید که چرا زحمت وارد کردن تابع عضو **execute** را که **virtual** محض می‌باشد به کلاس **Transaction** بخود داده‌ایم، در صورتیکه **execute** فاقد یک پیاده‌سازی غیرانتزاعی است. به لحاظ مفهومی ما این تابع عضو را وارد کرده‌ایم، به این دلیل که این تابع تعریف کننده رفتاری از تمام تراکنش‌ها یعنی اجرا شدن است. به لحاظ تکنیکی، بایستی تابع عضو **execute** را در کلاس مبنای **Transaction** قرار دهیم، برای اینکه

ATM (یا هر کلاس دیگری) بتواند بصورت چندریختی نسخه override شده این تابع را از طریق اشاره‌گر با مراجعه **Transaction** فراخوانی کند.

کلاس‌های مشتق شده **BalanceInquiry**، **Withdrawal** و **Deposit** صفت **accountNumber** را از کلاس مبنای **Transaction** به ارث می‌برند، اما کلاس‌های **Withdrawal** و **Deposit** حاوی صفت دیگری بنام **amount** هستند که وجه تمایز آنها از کلاس **BalanceInquiry** است. کلاس‌های **Widthdrawal** و **Deposit** نیازمند این صفت اضافی هستند تا بتوانند میزان پولی که کاربر می‌خواهد از حساب برداشت کند یا پس‌انداز نماید، ذخیره کنند. کلاس **BalanceInqiury** نیازی به این صفت ندارد و فقط نیازمند یک شماره حساب برای اجرا شدن دارد. با اینکه دو تا از سه کلاس مشتق شده **Transaction** این صفت را به اشتراک می‌گذارند، ما آنرا در کلاس مبنا **Transaction** جای نداده‌ایم. فقط ویژگیهای مشترک در میان تمام کلاس‌های مشتق شده را در کلاس مبنا جای می‌دهیم، از اینرو کلاس‌های مشتق شده صفات غیرضروری (و عملیات) را به ارث نمی‌برند.

شکل ۲۸-۱۳ تصویری از دیاگرام کلاس به روز شده از مدلی است که توارث در آن وارد و به معرفی کلاس **Transaction** پرداخته شده است. یک پیوستگی مابین کلاس **ATM** و کلاس **Transaction** را مدل کرده‌ایم تا نشان دهیم که در هر لحظه، تراکنشی در ATM اجرا می‌شود یا خیر (یعنی، صفر یا یک شی از نوع **Transaction** در یک زمان در سیستم وجود دارد).

شکل ۲۸-۱۳ | دیاگرام کلاس از سیستم ATM (پیوستگی توارث). دقت کنید که نام کلاس انتزاعی Transaction بصورت ایتالیک نوشته شده است.

بدلیل اینکه **Withdrawal** نوعی **Transaction** است، نیازی به ترسیم خط پیوستگی مستقیماً مابین کلاس **ATM** و کلاس **Withdrawal** نداریم. کلاس‌های مشتق شده **BalanceInquiry** و **Deposit** نیز این پیوستگی را به ارث می‌برند.

همچنین یک پیوستگی مابین کلاس **Transaction** و **BankDatabase** اضافه کرده‌ایم (شکل ۲۸-۱۳). تمام تراکنش‌ها نیازمند داشتن یک مراجعه به **BankDatabase** هستند. از اینرو می‌توانند به اطلاعات حساب دسترسی پیدا کنند. هر کلاس مشتق شده از **Transaction** این مراجعه را به ارث می‌برد و بنابر این پیوستگی موجود مابین کلاس **Withdrawal** و **BankDatabase** را مدل نکرده‌ایم.

یک پیوستگی مابین کلاس **Transaction** و **Screen** ایجاد کرده‌ایم به این دلیل که تمام تراکنش در خروجی و در دید کاربر از طریق **Screen** قرار می‌گیرند. هر کلاس مشتق شده‌ای این پیوستگی را به ارث می‌برد. کلاس **Withdrawal** هنوز هم سهیم در پیوستگی با **CashDispenser** و **Keypad** است، این پیوستگی‌ها بر روی کلاس مشتق شده از **Withdrawal** اعمال می‌شود.

دیاگرام کلاس به نمایش درآمده در شکل ۲۰-۹ نشان دهنده صفات و عملیات با نشانگر قابل رویت بودن است. اکنون دیاگرام اصلاح شده کلاس را در شکل ۲۹-۱۳ عرضه کرده‌ایم که شامل کلاس مبنای انتزاعی **Trancaction** می‌باشد. این دیاگرام مختصر شده، رابطه توارث را نشان می‌دهد (این روابط در شکل ۲۸-۱۳ آورده شده‌اند)، اما بجای آن صفات و عملیات‌های که پس از اعمال ارث‌بری به سیستم بدست آمده‌اند را در خود دارد. توجه کنید که نام کلاس انتزاعی **Transaction** و نام عملیات انتزاعی **execute** در کلاس **Transaction** بصورت ایتالیک نشان داده شده است.

پیاده‌سازی طرح سیستم ATM با توارث

در بخش ۱۲-۹ شروع به پیاده‌سازی طرح سیستم ATM با کد C++ کردیم. در این بخش سعی می‌کنیم با اصلاح آن پیاده‌سازی از توارث سود ببریم و از کلاس **Withdrawal** بعنوان یک مثال استفاده می‌کنیم.

۱- اگر کلاس **A** تعمیم دهنده کلاس **B** باشد، پس کلاس **B** از کلاس **A** مشتق شده است. برای مثال، کلاس مبنای انتزاعی **Transaction** تعمیم دهنده کلاس **Withdrawal** است. از اینرو، کلاس **Withdrawal** از کلاس **Transaction** مشتق شده است. شکل ۳۰-۱۳ حاوی بخشی از فایل سرآیند کلاس **Withdrawal** می‌باشد، که در آن تعریف کلاس نشاندهنده رابطه ارث‌بری مابین **Withdrawal** و **Transaction** است (خط 9)

۲- اگر کلاس **A** یک کلاس انتزاعی باشد و کلاس **B** از کلاس **A** مشتق شده باشد، اگر کلاس **B** یک کلاس غیرانتزاعی باشد، پس کلاس **B** باید توابع **virtual** محض کلاس **A** را پیاده‌سازی کند. برای مثال، کلاس **Transaction** حاوی تابع **execute** است که **virtual** محض می‌باشد، از اینرو کلاس **Widthdrawal** باید این تابع عضو را پیاده‌سازی کند، اگر مایل باشیم نمونه‌ای از شی **Withdrawal** ایجاد کنیم. شکل ۳۱-۱۳ حاوی فایل سرآیند C++ برای کلاس **Withdrawal** از شکل ۲۸-۱۳ و شکل ۲۹-۱۳ می‌باشد. کلاس **Withdrawal** عضو داده **accountNumber** را از کلاس مبنای **Transaction** به ارث می‌برد، از اینرو **Withdrawal** این عضو داده را اعلان نکرده است. همچنین کلاس **Withdrawal** مراجعه‌های به **Screen** و **BankDatabase** را از کلاس مبنای خود یعنی **Transcation** به ارث برده است، بنابر این نیازی نیست تا این مراجعه‌ها را در کد خودمان وارد کنیم. شکل ۲۹-۱۳ نشاندهنده صفت **amount** و عملیات **execute** متعلق به کلاس **Withdrawal** است. خط 19 از شکل ۳۱-۱۳ یک عضو داده برای صفت **amount** اعلان کرده است. خط 16 حاوی نمونه اولیه تابع برای عملیات **execute** است. بخاطر داشته باشید که برای غیرانتزاعی بودن یک کلاس، کلاس مشتق شده **Withdrawal** بایستی یک پیاده‌سازی غیرانتزاعی از تابع **execute** که **virtual** محض در کلاس مبنای

Transaction می‌باشد تدارک دیده باشد. نمونه اولیه در خط 16 هشدار می‌دهد که تابع **virtual** محض در کلاس مبنا را override کنید.

اگر بخواهید یک پیاده‌سازی در فایل cpp. داشته باشید، باید این نمونه اولیه را تدارک ببینید. مراجعه‌های **keypad** و **cashDispenser** (خطوط 21-20) اعضای داده مشتق شده از پیوستگی **Withdrawal** در شکل ۲۸-۱۳ هستند. برای اینکه بتوانیم مراجعه‌های موجود در خطوط 21-20 را کامپایل کنیم از اعلان‌های رو به جلو در خطوط 9-8 استفاده کرده‌ایم.

```
1  // Fig. 13.30: Withdrawal.h
2  // Withdrawal class definition. Represents a withdrawal transaction.
3  #ifndef WITHDRAWAL_H
4  #define WITHDRAWAL_H
5
6  #include "Transaction.h" // Transaction class definition
7
8  //class withdrawal derives from base class Transaction
9  class Withdrawal : public Transaction
10 {
11 }; // end class Withdrawal
12
13 #endif // WITHDRAWAL_H
```

شکل ۳۰-۱۳ | تعریف کلاس Withdrawal که از Transaction مشتق شده است.

```
1  // Fig. 13.31: Withdrawal.h
2  // Definition of class Withdrawal that represents a withdrawal transaction.
3  #ifndef WITHDRAWAL_H
4  #define WITHDRAWAL_H
5
6  #include "Transaction.h" // Transaction class definition
7
8  class Keypad; // forward declaration of class Keypad
9  class CashDispenser; // forward declaration of class CashDispenser
10
11 //class withdrawal derives from base class Transaction
12 class Withdrawal : public Transaction
13 {
14 public:
15    //member function overriding execute in base class Transaction
16    virtual void execute(); // perform the transaction
17 private:
18    // attributes
19    double amount; // amount to withdraw
20    Keypad &keypad; // reference to ATM's keypad
21    CashDispenser &cashDispenser; // reference to ATM's cash dispenser
22 }; // end class Withdrawal
23
24 #endif // WITHDRAWAL_H
```

شکل ۳۱-۱۳ | فایل سرآیند کلاس Withdrawal مبتنی بر شکل‌های ۲۸-۱۳ و ۲۹-۱۳.

خودآزمایی مبحث مهندسی نرم‌افزار

۱-۱۳ زبان UML از یک فلش با..... برای نشان دادن رابطه تعمیم استفاده می‌کند.

a) فلش توپر.

b) فلش مثلثی توخالی.

c) فلش توخالی لوزی شکل.

d) فلش یکپارچه.

۱۳-۲ آیا عبارت زیر صحیح است یا اشتباه در زبان UML در زیر اسامی کلاس انتزاعی و عملیات یک خط قرار داده می‌شود.

۱۳-۳ یک فایل سرآیند C++ بنویسید تا پیاده‌سازی از طرح کلاس Transaction مشخص شده در شکل ۱۳-۲۸ و ۱۳-۲۹ باشد. مطمئن شوید تا مراجعه‌های private بر پایه پیوستگی کامل Transaction در نظر گرفته شده باشند. همچنین از توابع سراسری get برای اعضای داده private استفاده کنید تا کلاس‌های مشتق شده بتوانند وظایف خود را انجام دهند.

پاسخ خودآزمایی مبحث آموزشی مهندسی نرم‌افزار

۱۳-۱ b.

۱۳-۲ اشتباه. در UML اسامی و عملیات کلاس انتزاعی بصورت ایتالیک نشان داده می‌شود.

۱۳-۳ حاصل طراحی کلاس Transaction در فایل سرآیند شکل ۱۳-۲۲ آورده شده است.

```
// Fig. 13.32: Transaction.h
// Transaction abstract base class definition.
#ifndef TRANSACTION_H
#define TRANSACTION_H

class Screen; // forward declaration of class Screen
class BankDatabase; // forward declaration of class BankDatabase

class Transaction
{
public:
   int getAccountNumber() // return account number
   Screen &getScreen() // return reference to screen
   BankDatabase &getBankDatabase() // return reference to database

   // pure virtual function to perform the transaction
   virtual void execute() = 0; // overridden in derived classes
private:
   int accountNumber; // indicates account involved
   Screen &screen; // reference to the screen of the ATM
   BankDatabase &bankDatabase; // reference to the account info database
}; // end class Transaction

#endif // TRANSACTION_H
```

شکل ۱۳-۳۲ | فایل سرآیند کلاس Transaction بر پایه شکل‌های ۱۳-۲۸ و ۱۳-۲۹.

فصل
چهاردهم

الگوها

اهداف

- استفاده از الگوهای تابع در ایجاد راحتر توابع مرتبط.
- تمایز قائل شدن مابین الگوهای تابع و الگوهای تابع تخصصی شده.
- استفاده از الگوهای کلاس برای ایجاد گروهی از نوع‌های مرتبط.
- تمایز قائل شدن مابین الگوهای کلاس و الگوهای کلاس تخصصی شده.
- سربارگذاری الگوهای تابع.
- درک رابطه موجود مابین الگوها، دوستان، توارث و اعضای استاتیک.

رئوس مطالب

۱٤-۱ مقدمه

در این فصل، در ارتباط با یکی از ویژگیهای قدرتمند C++ که در استفاده مجدد از نرم‌افزار نقش دارد و بنام *الگوها* یا *قالب‌ها* شناخته می‌شود صحبت خواهیم کرد. الگوهای تابع و الگوهای کلاس به برنامه‌نویس امکان می‌دهند تا جنبه خاصی به یک بخش از کد، کل توابع مرتبط (که الگوهای تابع تخصصی شده نامیده می‌شوند) یا کل کلاس‌های مرتبط (که الگوهای کلاس تخصصی شده نامیده می‌شوند) اعطا کند. این تکنیک بنام *برنامه‌نویسی عمومی* شناخته می‌شود.

می‌توانیم یک الگوی تابع منفرد در یک تابع مرتب‌سازی آرایه بنویسیم، سپس با داشتن الگوی تابع تخصصی شده که توسط C++ تولید می‌شود می‌توان آرایه‌های از نوع int، float، string و غیره را مرتب کرد. در فصل ششم به معرفی الگوها پرداخته‌ایم. در این فصل مباحث دیگری به آن اضافه می‌کنیم.

می‌توانیم یک الگوی کلاس منفرد برای کلاس پشته بنویسیم، سپس همانند کلاس پشته int، کلاس پشته float، کلاس پشته string و غیره توسط C++ تولید می‌گردد.

باید به وجه تمایز مابین الگوها و الگوهای تخصصی شده توجه کرد: الگوهای تابع و الگوهای کلاس همانند شابلون یا استنسیل هستند که برای ترسیم استفاده می‌کنیم، الگوهای تابع تخصصی شده و الگوهای کلاس تخصصی شده همانند ترسیم‌های جداگانه‌ای هستند که تماماً همشکل بوده، اما می‌توانند برای مثال با رنگ‌های مختلفی ترسیم شوند.

در این فصل، به معرفی یک الگوی تابع و الگوی کلاس خواهیم پرداخت. همچنین به رابطه موجود مابین الگوها و سایر ویژگیهای C++ توجه می‌کنیم، ویژگیهای همانند سربارگذاری، توارث، دوستان و اعضای استاتیک. جزئیات و مکانیزم طراحی الگوها که در این فصل توضیح می‌دهیم براساس مقاله آقای Byarne Stroustrup بنام *Parameterized Types for C++* و انتشار یافته در کنفرانس *Proceedings of the USENIX C++* در دنور، کلورادو به سال 1998 است. این فصل خاص الگوها است.

۱۴-۲ الگوهای تابع

معمولاً سربارگذاری توابع عملیات مشابه یا یکسانی بر روی انواع مختلف داده انجام می‌دهد. اگر عملیات‌ها برای هر نوع یکسان باشند، می‌توان آنها را بسیار جمع و جورتر و مناسب‌تر و با استفاده از الگوهای تابع بیان کرد. در ابتدا، برنامه‌نویس یک تعریف منفرد از الگوی تابع می‌نویسد. بر پایه انواع آرگومان تدارک دیده شده بصورت صریح یا استنتاج شده از فراخوانی این تابع، کامپایلر مبادرت به تولید کد شی مجزا برای تابع می‌کند (یعنی الگوی تابع تخصصی شده) تا به فراخوانی هر تابع بطرز مناسبی پاسخ داده شود. در زبان C، اینکار با استفاده از ماکروها که با رهنمود **#define** تولید می‌شود، قابل انجام است. با این وجود، ماکروها می‌توانند تاثیرات جانبی خطرناکی را تولید کنند و به کامپایلر امکان بررسی نوع را نمی‌دهند. الگوهای تابع یک راه حل، کامل و فشرده همانند ماکروها عرضه می‌کنند اما از قابلیت بررسی نوع هم برخوردار هستند.

کلیه تعاریف الگوی تابع با کلمه کلیدی **template** و بدنبال آن یک لیست از پارامترهای الگو آغاز می‌شود که با کاراکترهای < و > احاطه می‌شوند. هم پارامتر الگو که نشاندهنده یک نوع است بایستی جلوتر از کلمات کلیدی **class** یا **typename** قرار داده شود، همانند

```
template< typename T >
```

یا

```
template< class ElementType >
```

یا

```
template< typename BorderType, typename FillType >
```

از پارامترهای نوع الگو که در تعریف الگوی تابع قرار دارند، برای مشخص کردن نوع آرگومان در تابع، نوع برگشتی تابع و اعلان متغیرهای موجود در درون تابع استفاده می‌شود. دقت کنید که از کلمات کلیدی **typename** و **class** برای مشخص کردن پارامترهای الگوی تابع استفاده شده است و در واقع به معنی «هر نوع توکار یا نوع تعریف شده توسط کاربر» است.

مثال: الگوی تابع printArray

اجازه دهید تا به بررسی الگوی تابع **printArray** در شکل ۱-۱۴، خطوط 15-8 بپردازیم. الگوی تابع **printArray** یک پارامتر الگو بنام **T** در خط 8 اعلان کرده است (**T** می‌تواند هر شناسه معتبری باشد) که برای نوع آرایه قابل چاپ توسط **printArray** است، از **T** بعنوان نوع پارامتر الگو یا پارامتر نوع یاد می‌شود. در بخش ۵-۱۴ با پارامترهای بدون نوع آشنا خواهید شد.

زمانی که کامپایلر احضار تابع **printArray** را در برنامه سرویس‌گیرنده (همانند خطوط 30, 35 و 40) تشخیص داد، با استفاده از قابلیت تفکیک سربارگذاری بهترین تعریف تابع **printArray** را پیدا می‌کند. در این مورد، تنها تابع **printArray** با تعداد مناسب پارامترها، الگوی تابع **printArray** است (خطوط 8-

15). به فراخوانی تابع در خط 30 توجه کنید. کامپایلر مبادرت به مقایسه نوع اولین آرگومان **printArray** (یعنی ***int*** در خط 30) با اولین پارامتر الگوی تابع **printArray** (یعنی ***const T*** در خط 9) کرده و استنباط می‌کند که نوع پارامتر **T** را با **int** جایگزین سازد تا آرگومان مطابق با پارامتر گردد. سپس کامپایلر **int** را جانشین **T** در کل تعریف الگو کرده و تابع تخصصی شده **printArray** را پردازش می‌کند که می‌تواند یک آرایه از مقادیر صحیح را به نمایش در آورد. در برنامه شکل ۱-۱۴، کامپایلر سه تابع تخصصی شده از **printArray** ایجاد می‌کند، یکی که در انتظار آرایه از نوع **int**، یکی برای آرایه از نوع **double** و دیگری برای آرایه از نوع **char** است. برای مثال، الگوی تابع تخصصی شده برای نوع **int** بصورت زیر است:

```
void printArray( const int *array, int count )
{
    for ( int i = 0; i < count; i++ )
        cout << array[i] << " ";
    cout << endl;
} //end function printArray
```

نام یک پارامتر الگو می‌تواند فقط یکبار در لیست پارامتری الگو در سرآیند الگو اعلان شود، اما می‌تواند به دفعات در سرآیند و بدنه تابع بکار گرفته شود. لازم نیست تا اسامی پارامتر الگو در میان الگوهای تابع منحصر بفرد باشد.

```
1  // Fig 14.1: fig14_01.cpp
2  // Using template functions.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  // function template printArray definition
8  template< typename T >
9  void printArray( const T *array, int count )
10 {
11    for ( int i = 0; i < count; i++ )
12       cout << array[ i ] << " ";
13
14    cout << endl;
15 } // end function template printArray
16
17 int main()
18 {
19    const int aCount = 5; // size of array a
20    const int bCount = 7; // size of array b
21    const int cCount = 6; // size of array c
22
23    int a[ aCount ] = { 1, 2, 3, 4, 5 };
24    double b[ bCount ] = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
25    char c[ cCount ] = "HELLO"; // 6th position for null
26
27    cout << "Array a contains:" << endl;
28
29    // call integer function-template specialization
30    printArray( a, aCount );
31
32    cout << "Array b contains:" << endl;
33
34    // call double function-template specialization
35    printArray( b, bCount );
36
```

```
37    cout << "Array c contains:" << endl;
38
39    // call character function-template specialization
40    printArray( c, cCount );
41    return 0;
42 } // end main
```

```
Array a contains:
1 2 3 4 5
Array b contains:
1.1 2.2 3.3 4.4 5.5 6.6 7.7
Array c contains:
H E L L O
```

شکل ۱-۱٤ | تخصصی کردن الگوی تابع printArray.

در شکل ۱-۱۴ به بررسی الگوی تابع **printArray** پرداخته شده است (خطوط 15-8). برنامه با اعلان پنج عنصر آرایه **a** از نوع **int**، هفت عنصر آرایه b از نوع **double** و شش عنصر آرایه **c** از نوع **char** آغاز می‌شود (به ترتیب خطوط 25-23). سپس برنامه با فراخوانی **printArray** مبادرت به چاپ هر آرایه می‌کند، یکبار با آرگومان اول **a** از نوع *** int** (خط 30)، یکبار با آرگومان اول **b** از نوع *** double** (خط 35) و یکبار با آرگومان اول **c** از نوع *** char** (خط 40). برای مثال، با فراخوانی موجود در خط 30، کامپایلر استنتاج می‌کند که **T** یک **int** بوده و نمونه تخصصی از تابع الگوی **printArray** ایجاد می‌کند که در آن نوع پارامتر **T**، صحیح **(int)** است. فراخوانی موجود در خط 35 سبب می‌شود که کامپایلر استنتاج کند که **T** یک **double** بوده و نمونه دوم تخصصی شده از تابع الگوی **printArray** را ایجاد می‌کند که در آن نوع پارامتر **T** از نوع **double** است. فراخوانی موجود در خط 40 سبب می‌شود که کامپایلر استنتاج کند که **T** از نوع **char** است و نمونه سومی از تابع تخصصی شده **printArray** ایجاد می‌کند که در آن **T** از نوع **char** می‌باشد. نکته مهم در اینجاست که اگر **T** (خط 8) نشاندهنده یک نوع تعریف شده توسط کاربر باشد (که در شکل ۱-۱۴ این چنین نیست)، بایستی مبادرت به سربارگذاری عملگر درج برای آن نوع می‌کردیم، در غیر اینصورت، اولین عملگر درج در خط 12 کامپایلر نمی‌شود.

در این مثال، مکانیزم الگو سبب می‌شود تا برنامه‌نویس مجبور به نوشتن سه تابع مجزای سربارگذاری شده با نمونه‌های اولیه زیر نشود:

```
void printArray( const int *,int );
void printArray( const double *,int );
void printArray( const char *,int );
```

که همگی از کد یکسانی بجز نوع T استفاده می‌کنند (همانطوری که در خط 9 بکار گرفته شده است).

۳-۱٤ سربارگذاری الگوهای تابع

الگوهای تابع و سربارگذاری با یکدیگر مرتبط هستند. الگوی تابع تخصصی شده از یک الگوی تابع تولید می‌شود که همگی دارای نام یکسان هستند، از اینرو کامپایلر با استفاده از تفکیک‌پذیری سربارگذاری مبادرت به فراخوانی تابع مناسب می‌کند.

به چندین روش می‌توان یک الگوی تابع را سربارگذاری کرد. می‌توانیم الگوهای تابع دیگر تدارک ببینیم که دارای نام مشابه بوده اما در پارامترهای تابع با هم تفاوت داشته باشد. برای مثال، الگوی تابع **printArray** در شکل ۱-۱۴ می‌توانست با الگوی تابع دیگر **printArray** با پارامترهای اضافی **lowSubscript** و **highSubscript** که تعیین می‌کنند چه محدوده‌ای در آرایه چاپ شود، سربارگذاری گردد.

همچنین یک الگوی تابع می‌تواند با تدارک دیدن توابع غیر الگو با نام تابع مشابه اما متفاوت در آرگومان‌های تابع، سربارگذاری گردد. برای مثال، الگوی تابع **printArray** در شکل ۱-۱۴ می‌توانست با یک نسخه غیر الگو که خاص چاپ آرایه‌ای از رشته‌های کاراکتری بصورت مرتب با فرمت جدولی است، سربارگذاری شود.

کامپایلر با استفاده از یک روش تطبیق دهنده تعیین می‌کند که کدام تابع به هنگام احضار، فراخوانی گردد. ابتدا، کامپایلر تمام الگوهای تابع را که مطابق با نام تابع فراخوانی شده هستند، را یافته و بر پایه آرگومان‌های موجود در تابع فراخوانی شده، مبادرت به ایجاد توابع تخصصی شده می‌کند. سپس کامپایلر تمام توابع متداول را که مطابق با نام برده شده باشند، پیدا می‌کند. اگر یکی از توابع متداول یا الگوی تابع تخصصی شده بهترین مطابقت را با تابع فراخوانی شده داشته باشد، آن تابع یا تابع تخصصی شده بکار گرفته می‌شود. اگر هر دو تابع به یک میزان مطابقت داشته باشند، تابع متداول یا عادی انتخاب خواهد شد. اگر چندین مطابقت برای فراخوانی یک تابع رخ دهد، کامپایلر دچار ابهام شده و پیغام خطا صادر می‌کند.

۱۴-۴ الگوهای کلاس

درک مفهوم پشته مستقل از نوع آیتم‌های است که به آن وارد می‌شوند. با این وجود، در نمونه‌سازی یک پشته، بایستی نوع داده مشخص شود. چنین حالتی فرصت مناسبی برای بهره‌مند شدن از ویژگی استفاده مجدد از نرم‌افزار است. تنها چیزی که نیاز داریم توجه به اصل پشته و ایجاد کلاس‌های است که مرتبط با نوع‌های مختلف می‌باشند و با پشته کار می‌کنند. C++ این قابلیت را از طریق الگوهای کلاس فراهم آورده است.

الگوهای کلاس معروف به نوع‌های پارامتری شده هستند، چرا که آنها مستلزم یک یا چندین نوع پارامتر برای مشخص کردن نحوه بهینه‌سازی یک الگوی «کلاس کلی» بفرم یک الگوی کلاس تخصصی شده می‌باشند.

برنامه‌نویسی که مایل به تهیه نسخه‌های متفاوتی از الگوی کلاس تخصصی شده است، فقط یک تعریف از الگوی کلاس را کدنویسی می‌کند. هر بار که به یک الگوی کلاس تخصصی شده دیگر نیاز پیدا شود، برنامه‌نویس از عبارات بسیار مختصر و فشرده استفاده کرده و کامپایلر کد مورد نیاز برای آن را تولید

می‌کند. برای مثال، الگوی کلاس **Stack**، می‌تواند تبدیل به پایه‌ای برای ایجاد کلاس‌های متعدد **Stack** (همانند "پشته‌ای از مقادیر **double**"، "پشته‌ای از مقادیر **int**"، "پشته‌ای از مقادیر **char**"، "پشته‌ای از کارمندان" و غیره) در برنامه شود.

ایجاد الگوی کلاس Stack <T>

به تعریف الگوی کلاس **Stack** (پشته) در شکل ۲-۱۴ توجه کنید. ظاهر آن شبیه تعریف یک کلاس عادی می‌باشد، بجز سرآیند قرار گرفته در خط 6

```
Template< typename T >
```

عبارت فوق تصریح کننده تعریف الگوی کلاس با پارامتر نوع **T** است که همانند یک جانگهدار برای نوع کلاس **Stack** که ایجاد خواهد شد عمل می‌کند. نیازی نیست که برنامه‌نویس حتماً از شناسه **T** استفاده کند و می‌تواند از هر شناسه معتبر دیگری استفاده نماید. نوع عنصر ذخیره شده در این پشته در سرتاسر سرآیند کلاس **T** و تعریف توابع عضو بعنوان **T** شناخته می‌شود. همانطوری که خواهید دید، **T** می‌تواند به نوع مشخصی همانند **int** یا **double** تبدیل شود. با توجه به روش طراحی این الگوی کلاس، دو محدودیت برای نوع داده‌های غیربنیادین بکار رفته در این پشته در نظر گرفته شده است، آنها باید یک سازنده پیش‌فرض (برای استفاده در خط 44 به منظور ایجاد آرایه‌ای که عناصر پشته را ذخیره می‌کند) داشته باشند و بایستی از عملگر تخصیص پشتیبانی کنند (خط 55 و 69). تعریف تابع عضو از یک الگوی کلاس، الگوهای تابع هستند. تعریف تابع عضو که خارج از تعریف الگوی کلاس جای می‌گیرد، با سرآیند زیر آغاز می‌گردد:

```
template< typename T >
```

(خطوط 40, 51 و 65). از اینرو، هر تعریفی شباهت به تعریف یک تابع عادی دارد، بجز اینکه نوع عنصر پشته همیشه از نوع پارامتر **T** خواهد بود. از عملگر باینری تفکیک قلمرو به همراه نام الگوی کلاس **Stack<T>** بکار گرفته شده (خطوط 41، 52 و 66) تا تعریف تابع عضو به قلمرو الگوی کلاس پیوند زده شود. در این مورد، نام کلاس عمومی **Stack<T>** است. زمانیکه پشته **doubleStack** بصورت نوع **Stack<double>** نمونه‌سازی می‌شود، سازنده **Stack** الگوی تابع تخصصی شده از **new** برای ایجاد آرایه‌ای از عناصر نوع **double** برای عرضه پشته استفاده کرده است (خط 44). عبارت

```
stackPtr = new T[ size ];
```

در تعریف الگوی کلاس **Stack** توسط کامپایلر در الگوی کلاس در الگوی کلاس تخصصی شده **Stack<double>** بصورت زیر تولید می‌شود

```
stackPtr = new double[ size ];
```

```
1  // Fig. 14.2: Stack.h
2  // Stack class template.
3  #ifndef STACK_H
4  #define STACK_H
5
6  template< typename T >
```

```
7  class Stack
8  {
9  public:
10    Stack( int = 10 ); // default constructor (Stack size 10)
11
12    // destructor
13    ~Stack()
14    {
15       delete [] stackPtr; // deallocate internal space for Stack
16    } // end ~Stack destructor
17
18    bool push( const T& ); // push an element onto the Stack
19    bool pop( T& ); // pop an element off the Stack
20
21    // determine whether Stack is empty
22    bool isEmpty() const
23    {
24       return top == -1;
25    } // end function isEmpty
26
27    // determine whether Stack is full
28    bool isFull() const
29    {
30       return top == size - 1;
31    } // end function isFull
32
33 private:
34    int size; // # of elements in the stack
35    int top; // location of the top element (-1 means empty)
36    T *stackPtr; // pointer to internal representation of the Stack
37 }; // end class template Stack
38
39 // constructor template
40 template< typename T >
41 Stack< T >::Stack( int s )
42    : size( s > 0 ? s : 10 ), // validate size
43      top( -1 ), // Stack initially empty
44      stackPtr( new T[ size ] ) // allocate memory for elements
45 {
46    // empty body
47 } // end Stack constructor template
48
49 // push element onto Stack;
50 // if successful, return true; otherwise, return false
51 template< typename T >
52 bool Stack< T >::push( const T &pushValue )
53 {
54    if ( !isFull() )
55    {
56       stackPtr[ ++top ] = pushValue; // place item on Stack
57       return true; // push successful
58    } // end if
59
60    return false; // push unsuccessful
61 } // end function template push
62
63 // pop element off Stack;
64 // if successful, return true; otherwise, return false
65 template< typename T >
66 bool Stack< T >::pop( T &popValue )
67 {
68    if ( !isEmpty() )
69    {
70       popValue = stackPtr[ top-- ]; // remove item from Stack
71       return true; // pop successful
72    } // end if
73
74    return false; // pop unsuccessful
75 } // end function template pop
76
```

```
77 #endif
```

شکل ۲-۱٤ | الگوی کلاس Stack.

ایجاد یک راه‌انداز برای تست الگوی کلاس Stack<T>

حال اجازه دهید تا به بررسی راه‌اندازی بپردازیم (شکل ۳-۱۴) که الگوی کلاس **Stack** را بکار می‌گیرد. راه‌انداز کار را با نمونه‌سازی شی **doubleStack** با سایز 5 آغاز می‌کند (خط 11). این شی بصورت کلاس **Stack<double>** اعلان می‌شود. کامپایلر نوع **double** را با پارامتر نوع **T** در الگوی کلاس به منظور تولید کد منبع کلاس **Stack** از نوع **double** پیوند می‌دهد. اگرچه الگوها ارائه کننده مزیت استفاده مجدد از نرم‌افزار هستند اما بخاطر داشته باشید الگوی کلاس تخصصی شده مضاعفی در یک برنامه نمونه‌سازی می‌گردد (در زمان کامپایل)، با اینکه الگو فقط یکبار نوشته می‌شود.

خطوط 17-21 مبادرت به احضار تابع **push** برای قرار دادن مقادیر 1.1، 2.2، 3.3، 4.4 و 5.5 از نوع **double** به **doubleStack** می‌کنند. حلقه **while** زمانی خاتمه می‌یابد که راه‌انداز مبادرت به وارد کردن ششمین مقدار به **doubleStack** کند (حالتی که پشته پر است، چرا که حداکثر عنصری که می‌تواند این پشته نگهداری کند، پنج عنصر است). دقت کنید که تابع **push** زمانیکه قادر به وارد کردن مقداری به پشته نباشد، false برگشت می‌دهد.

خطوط 27-28 تابع **pop** را در حلقه **while** فراخوانی می‌کنند تا پنج مقدار از پشته خارج شود یا حذف گردد. زمانیکه راه‌انداز مبادرت به خارج کردن ششمین مقدار از پشته نماید و با توجه به اینکه **doubleStack** خالی است، حلقه خاتمه می‌یابد.

```
1  // Fig. 14.3: fig14_03.cpp
2  // Stack class template test program.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Stack.h" // Stack class template definition
8
9  int main()
10 {
11    Stack< double > doubleStack( 5 ); // size 5
12    double doubleValue = 1.1;
13
14    cout << "Pushing elements onto doubleStack\n";
15
16    // push 5 doubles onto doubleStack
17    while ( doubleStack.push( doubleValue ) )
18    {
19       cout << doubleValue << ' ';
20       doubleValue += 1.1;
21    } // end while
22
23    cout << "\nStack is full. Cannot push " << doubleValue
24       << "\n\nPopping elements from doubleStack\n";
25
26    // pop elements from doubleStack
27    while ( doubleStack.pop( doubleValue ) )
28       cout << doubleValue << ' ';
29
30    cout << "\nStack is empty. Cannot pop\n";
```

```
31
32    Stack< int > intStack; // default size 10
33    int intValue = 1;
34    cout << "\nPushing elements onto intStack\n";
35
36    // push 10 integers onto intStack
37    while ( intStack.push( intValue ) )
38    {
39       cout << intValue << ' ';
40       intValue++;
41    } // end while
42
43    cout << "\nStack is full. Cannot push " << intValue
44       << "\n\nPopping elements from intStack\n";
45
46    // pop elements from intStack
47    while ( intStack.pop( intValue ) )
48       cout << intValue << ' ';
49
50    cout << "\nStack is empty. Cannot pop" << endl;
51    return 0;
52 } // end main
```

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Stack is full. Cannot push 6.6

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Stack is empty. Cannot pop

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10
Stack is full. Cannot push 11

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty. Cannot pop
```

شکل ۳-۱٤ | برنامه تست الگوی کلاس Stack.

خط 32 مبادرت به نمونه‌سازی پشته **intStack** با اعلان زیر می‌کند

```
Stack< int > intStack;
```

چون سایزی مشخص نشده است، سایز پیش‌فرض 10 برای سازنده پیش‌فرض در نظر گرفته می‌شود (خط 10 از شکل ۲-۱۴). حلقه موجود در خطوط 41-37 و احضار تابع **push** سبب می‌شود تا مقادیر وارد **inStack** شوند تا زمانیکه پشته پر گردد. سپس حلقه خطوط 48-47 و احضار تابع **pop** سبب می‌شود تا مقادیر از intStack تا خالی شدن پشته ادامه یابد.

ایجاد الگوهای تابع برای تست الگوهای کلاس <Stack<T>

اگر دقت کنید متوجه می‌شوید کد موجود در تابع **main** شکل ۳-۱۴ تقریباً با **doubleStack** در خطوط 11-30 و **intStack** در خطوط 50-32 یکسان است. اینحالت فرصت دیگری برای استفاده از یک الگوی تابع را فراهم می‌آورد. در برنامه شکل ۴-۱۴ الگوی تابع **testStack** تعریف شده (خطوط 38-14) تا همان وظایف **main** در شکل ۳-۱۴ را انجام دهد. یعنی وارد کردن مقادیری به **Stack<T>** و خارج کردن مقادیر از **Stack<T>**. الگوی تابع **testStack** از پارامتر الگوی **T** (مشخص شده در خط 14) برای ارائه نوع داده ذخیره شده در **Stack<T>** استفاده کرده است. الگوی تابع، چهار آرگومان دریافت می‌کند

(خطوط 19-6)، یک مراجعه به یک شی از نوع **Stack<T>**، یک مقدار از نوع **T** که بعنوان اولین مقدار وارد پشته خواهد شد (push)، یک از نوع **T** برای افزایش مقادیر وارد شده به پشته و یک رشته که نشاندهنده نام پشته است که در خروجی از آن استفاده می‌شود. تابع **main** در خطوط 49-40 یک شی از نوع **Stack<double>** بنام **doubleStack** (خط 42) و یک شی از نوع **Stack<int>** بنام **intStack** (خط 43) ایجاد کرده و از این شی‌ها در خطوط 45 و 46 استفاده می‌کند. تابع **testStack** نتیجه کار را به نمایش در می‌آورد. کامپایلر نوع **T** برای **testStack** را از نوع بکار رفته برای نمونه‌سازی اولین آرگومان تابع استنتاج می‌کند. (یعنی نوع بکار رفته در نمونه‌سازی **doubleStasck** یا **intStack**). خروجی شکل ۴-۱۴ دقیقاً با خروجی شکل ۳-۱۴ مطابقت می‌کند.

```
1   // Fig. 14.4: fig14_04.cpp
2   // Stack class template test program. Function main uses a
3   // function template to manipulate objects of type Stack< T >.
4   #include <iostream>
5   using std::cout;
6   using std::endl;
7
8   #include <string>
9   using std::string;
10
11  #include "Stack.h" // Stack class template definition
12
13  // function template to manipulate Stack< T >
14  template< typename T >
15  void testStack(
16     Stack< T > &theStack, // reference to Stack< T >
17     T value, // initial value to push
18     T increment, // increment for subsequent values
19     const string stackName ) // name of the Stack< T > object
20  {
21     cout << "\nPushing elements onto " << stackName << '\n';
22
23     // push element onto Stack
24     while ( theStack.push( value ) )
25     {
26        cout << value << ' ';
27        value += increment;
28     } // end while
29
30     cout << "\nStack is full. Cannot push " << value
31        << "\n\nPopping elements from " << stackName << '\n';
32
33     // pop elements from Stack
34     while ( theStack.pop( value ) )
35        cout << value << ' ';
36
37     cout << "\nStack is empty. Cannot pop" << endl;
38  } // end function template testStack
39
40  int main()
41  {
42     Stack< double > doubleStack( 5 ); // size 5
43     Stack< int > intStack; // default size 10
44
45     testStack( doubleStack, 1.1, 1.1, "doubleStack" );
46     testStack( intStack, 1, 1, "intStack" );
47
48     return 0;
49  } // end main
```

```
Pushing elements onto doubleStack
```

```
1.2 2.2 3.3 4.4 5.5
Stack is full. Cannot push 6.6

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Stack is empty. Cannot pop

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10
Stack is full. Cannot push 11

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty. Cannot pop
```

شکل ٤-١٤ | ارسال یک شی الگوی Stack به یک الگوی تابع.

١٤-٥ پارامترهای بدون نوع و نوع‌های پیش‌فرض در الگوهای کلاس

در الگوی کلاس **Stack** از بخش ۴-۱۴، فقط از یک نوع پارامتر در سرآیند الگو استفاده شده بود (خط 6). البته امکان استفاده از پارامترهای الگوی بدون نوع یا پارامتر بدون نوع وجود دارد که می‌توانند آرگومان‌ها پیش‌فرض داشته باشند و با آنها همانند ثابت‌ها رفتار کرد. برای مثال، سرآیند الگو می‌تواند برای دریافت پارامتر **elements** از نوع **int** بصورت زیر تغییر یابد:

```
template< typename T, int elements > //nontype parameter elements
```

سپس، اعلانی نظیر زیر انجام داد

```
Stack< double, 100 > mostRecentSalesFigures;
```

که برای نمونه‌سازی (در زمان کامپایل) یک الگوی کلاس تخصصی شده **Stack** با 100 عنصر از مقادیر **double** بنام **mostRecentSalesFigures** بکار گرفته شود. سرآیند کلاس می‌توانست حاوی یک عضو داده **private** (خصوصی) با اعلان آرایه‌ای همانند زیر باشد

```
T stackHolderp[ elements ]; //array to hold Stack contents
```

علاوه بر این، یک پارامتر نوع می‌تواند بصورت یک نوع پیش‌فرض تعیین شود. برای مثال.

```
Template< typename T=string > //defaults to type string
```

مشخص می‌کند که **Stack** حاوی شی‌های رشته بصورت پیش‌فرض است. سپس، اعلانی همانند

```
Stack<> jobDescriptions;
```

می‌تواند برای نمونه‌سازی یک الگوی کلاس تخصصی شده **Stack** حاوی رشته‌ای بنام **jobDescriptions** بکار گرفته شود. این الگوی کلاس تخصصی شده می‌توانست از نوع **Stack<string>** باشد. پارامترها از نوع پیش‌فرض باید سمت راسترین پارامترها در لیست نوع پارامترها در الگو باشند.

در برخی از موارد، امکان استفاده از یک نوع خاص در یک الگوی کلاس وجود ندارد. برای مثال، الگوی **Stack** در شکل ۲-۱۴ مستلزم این است که نوع‌های تعریف شده توسط کاربر که در یک پشته ذخیره خواهند شد، بایستی یک سازنده پیش‌فرض و یک عملگر تخصیص در اختیار داشته باشد. اگر نوع خاص تعریف شده توسط کاربر با الگوی **Stack** ما کار نکند یا نیاز به بهینه‌سازی داشته باشد، می‌توانید یک

الگوی کلاس تخصصی شده صریح برای آن نوع خاص تعریف کنید. اجازه دهید فرض کنیم که می‌خواهیم یک **Stack** تخصصی صریح برای شی‌های **Employee** ایجاد کنیم. برای انجام اینکار، از یک کلاس جدید بنام **Stack<Employee>** بصورت زیر استفاده می‌کنیم:

```
template<>
class Stack< Employee >
{
   //body of class definition
};
```

توجه کنید که **Stack<Employee>** تخصصی شده صریح بطور کامل جایگزین الگوی کلاس **Stack** می‌شود که خاص نوع **Employee** است.

۱۴-۶ الگوها و توارث

الگوها و توارث در چند مورد با هم مرتبط هستند:

- یک الگوی کلاس می‌تواند از یک الگوی کلاس تخصصی شده مشتق گردد.
- یک الگوی کلاس می‌تواند از یک کلاس غیرالگو مشتق شود.
- یک الگوی کلاس تخصصی شده می‌تواند از یک الگوی کلاس تخصصی شده، مشتق شود.
- یک کلاس غیرالگو می‌تواند از یک الگوی کلاس تخصصی شده، مشتق شود.

۱۴-۷ الگوها و دوستان

مشاهده کردید که توابع و کل کلاس‌ها می‌توانند بعنوان دوستان **(friend)** کلاس‌های غیرالگو باشند. در الگوهای کلاس، رابطه دوستی می‌تواند مابین یک الگوی کلاس و یک تابع سراسری، یک تابع عضو از کلاس دیگری (که می‌تواند یک الگوی کلاس تخصصی شده نیز باشد) یا حتی کل یک کلاس برقرار گردد.

در سرتاسر این بخش، فرض ما بر این است که یک الگوی کلاس برای کلاس بنام **X** با پارامتر نوع **T** بصورت زیر تعریف کرده‌ایم:

```
template< typename T > class X
```

با توجه به این فرض، می‌توان **f1** را بعنوان دوست هر کلاس تخصصی شده از الگوی کلاس برای کلاس **X** ایجاد کرد. برای انجام اینکار، از اعلان رابطه دوستی بصورت زیر استفاده می‌کنیم

```
friend void f1();
```

برای مثال، تابع **f1** دوست **X<double>**، **X<string>** و **X<Employee>** و غیره است. همچنین امکان ایجاد تابع **f2** به نحوی که فقط دوست الگوی کلاس تخصصی شده باشد با همان نوع آرگومان وجود دارد. برای انجام اینکار، از اعلان رابطه دوستی بفرم زیر استفاده می‌شود

```
friend void f2( X< T > & );
```

برای مثال، اگر **T** یک **float** باشد، تابع **f2(x<float> &)** دوستی از الگوی کلاس تخصصی شده **X<float>** است اما دوست الگوی کلاس تخصصی شده **X<string>** نمی باشد.

می‌توانید یک تابع عضو یک کلاس دیگر را بعنوان دوست هر الگوی تخصصی تولید شده از الگوی کلاس اعلان کنید. برای انجام اینکار، اعلان **friend** باید تعیین کننده نام تابع عضو کلاس دیگر با استفاده از نام کلاس و عملگر باینری تفکیک قلمرو، باشد همانند

```
friend void A::f3();
```

این اعلان تابع عضو **f3** از کلاس **A** را دوست هر الگوی کلاس تخصصی شده و ایجاد شده از الگوی قبلی می‌کند. برای مثال تابع **f3** از کلاس **A** دوستی از **X<double>**، **X<string>** و **X<Employe>** و غیره می‌باشد.

همانند یک تابع سراسری، تابع عضو کلاس دیگر می‌تواند فقط دوست یک الگوی کلاس تخصصی شده با همان نوع آرگومان باشد. اعلان دوستی بفرم

```
friend void C< T >::f4( X< T > & );
```

برای نوع خاص **T** همانند **float** تابع عضو بصورت زیر ایجاد می‌شود

```
C< float >::f4( X< float > & )
```

یک تابع دوست فقط برای الگوی کلاس تخصصی شده **X<float>** است.

۸-۱٤ الگوها و اعضای استاتیک

نظرتان در مورد اعضای داده استاتیک چیست؟ بخاطر داشته باشید که با یک کلاس غیرالگو، یک کپی از هر عضو داده استاتیک در میان تمام شی‌های کلاس به اشتراک گذاشته می‌شود، و عضو داده استاتیک بایستی در قلمرو فایل اعلان شود.

هر الگوی کلاس تخصصی شده که از یک الگوی کلاس نمونه‌سازی شده است دارای یک کپی از هر عضو داده استاتیک از الگوی کلاس خواهد بود، تمام شی‌ها از آن الگوها تخصصی شده یک عضو داده استاتیکی را به اشتراک می‌گذارند. علاوه بر این، همانند اعضای داده استاتیک از کلاس‌های غیرالگو، اعضای داده از الگوی کلاس تخصصی شده بایستی تعریف شده و ضرورتاً در قلمرو فایل مقداردهی اولیه شود. هر الگوی تخصصی شده کپی متعلق بخود را از توابع عضو استاتیک الگوی کلاس بدست می‌آورد.

فصل
پانزدهم

استریم ورودی/خروجی

اهداف

- استفاده از استریم ورودی/خروجی شی‌گرا در C++.
- قالب‌بندی ورودی و خروجی.
- سلسله مراتب کلاس استریم I/O.
- استفاده از دستکاری کننده‌های استریم.
- کنترل ترازبندی و لایه‌گذاری.
- تعیین موفقیت‌آمیز بودن عملیات ورودی/خروجی.
- پیوند استریم ورودی به استریم خروجی.

رئوس مطالب

۱-۱۵ مقدمه

کتابخانه استاندارد C++ مجموعه وسیعی از قالبیت‌های ورودی/خروجی (I/O) را فراهم آورده است. در این فصل به بررسی قابلیت‌ها و توانایی عملیات I/O خواهیم پرداخت. C++ از I/O نوع ایمن (type-

(safe استفاد می‌کند. هر عملیات I/O به نوع داده حساس می‌باشد. اگر یک تابع عضو I/O برای کار با نوع داده خاصی در نظر گرفته شده باشد، فقط برای آن نوع داده فراخوانی می‌شود. اگر مطابقتی مابین نوع داده واقعی و تابع برای کار با آن نوع داده وجود نداشته باشد، کامپایلر خطا تولید خواهد کرد. از اینرو داده اشتباه قادر به نفوذ به سیستم نخواهد بود.

کاربران می‌توانند نحوه عملکرد I/O بر روی شی‌ها از نوع تعریف شده توسط کاربر را با اعمال سربارگذاری عملگرهای درج (>>) و استخراج (<<) مشخص سازند. این بسط‌پذیری یکی از ویژگیهای با ارزش C++ است.

۱۵-۲ استریم‌ها

در C++ عملیات I/O از طریق *استریم‌ها* (*streams*) یا *جریان‌ها* صورت می‌گیرد، که دنباله یا توالی از بایت‌ها هستند. در عملیات ورودی، جریان بایت‌ها از سوی یک دستگاه (همانند صفحه کلید، دیسک، اتصال شبکه) به حافظه اصلی است. در عملیات خروجی، جریان بایت‌ها از طرف حافظه اصلی به سمت یک دستگاه می‌باشد (همانند، صفحه نمایش، چاپگر، دیسک، اتصال شبکه و غیره). یک برنامه کاربردی با مفهوم پیوندی این بایت‌ها سر و کار دارد. بایت‌ها می‌توانند نشاندهنده کاراکترها، داده‌های خام، تصاویر گرافیکی، گفتارهای دیجیتالی، ویدئو دیجیتالی یا هر نوع اطلاعات دیگری باشند که مورد نیاز یک برنامه هستند.

مکانیزم سیستم I/O بایستی بایت‌ها را از دستگاه‌ها، حافظه بطور پایدار و قابل اعتماد انتقال دهد (و برعکس). غالباً چنین انتقالی مستلزم برخی اعمال و حرکات مکانیکی نظیر چرخش دیسک یا نوار یا تایپ به وسیله صفحه‌کلید است. زمانی که این نوع انتقال‌ها صرف می‌کنند به نسبت زمان مورد نیاز پردازنده برای کار بر روی داده‌های داخلی بیشتر است. از اینرو عملیات I/O مستلزم طرح دقیق و بهینه شده است تا از کارایی مناسب برخوردار باشد.

زبان C++ هر دو قابلیت I/O در «سطح پایین» و «سطح بالا» را تدارک دیده است. قابلیت I/O در سطح پایین (یعنی I/O قالب‌بندی نشده) مشخص می‌کند که تعدادی از بایت‌ها بایستی از دستگاه به حافظه یا از حافظه به دستگاه منتقل شوند. در چنین انتقالی، خود بایت آیتم مورد نظر و علاقه است. در I/O سطح پایین، سرعت و حجم انتقال بالا است اما مناسب برای برنامه‌نویسان نمی‌باشد.

معمولاً برنامه‌نویسان ترجیح می‌دهند از I/O سطح بالا (یعنی I/O قالب‌بندی شده) استفاده کنند که در آن بایت‌ها در واحدهای با معنی دسته‌بندی می‌شوند، همانند اعداد صحیح، اعشاری، کاراکترها، رشته و نوع‌های تعریف شده توسط کاربر. این نوع از عملیات I/O بسیار رضایت بخش‌تر از پردازش فایل با حجم بالا است.

۱-۲-۱۵ استریم‌های کلاسیک در مقابل استریم‌های استاندارد

در گذشته، کتابخانه استریم کلاسیک C++ قادر به انجام ورودی و خروجی بر روی کاراکترها (chars) بود. بدلیل اینکه یک **char** یک بایت فضا اشغال می‌کرد، فقط می‌توانست محدود به عرضه مجموعه‌ای از کاراکترها باشد (همانند کاراکترهای جدول ASCII). با این وجود، بسیاری از زبان‌ها از الفبا استفاده می‌کنند که حاوی کاراکترهای بیشتری به نسبت کاراکترهای قابل عرضه توسط یک char یک بایتی است. مجموعه کاراکتر ASCII نمی‌تواند این کاراکترها را تدارک ببیند، در حالیکه مجموعه کاراکتری Unicode قادر به انجام اینکار است. مجموعه Unicode یک مجموعه کاراکتری بسط یافته بین‌المللی است که حاوی بسیاری از حروف زبان‌های زنده و پرکاربرد، نمادهای محاسباتی و غیره است که در جهان کاربرد دارند. برای کسب اطلاعات بیشتر در مورد Unicode می‌توانید به www.unicode.org مراجعه کنید.

زبان C++ حاوی کتابخانه استریم استاندارد است که به برنامه‌نویسان امکان ایجاد سیستم‌های با قابلیت کار با کاراکترهای Unicode را فراهم می‌آورد. به همین منظور، C++ شامل یک نوع کاراکتر اضافی بنام **wchar_t** است که می‌تواند کاراکترهای Unicode را ذخیره کند. همچنین C++ استاندارد برای کار با کلاس‌های استریم کلاسیک C++ مجدداً طراحی شده است.

۲-۲-۱۵ فایل‌های سرآیند کتابخانه iostream

کتابخانه **iostream** در C++ حاوی صدها گزینه در ارتباط با I/O است. چندین فایل سرآیند حاوی بخش‌های از واسط کتابخانه هستند. اکثر برنامه‌های C++ شامل فایل سرآیند <**iostream**> می‌باشند که سرویس‌های پایه را برای تمام عملیات I/O اعلان می‌کند. فایل سرآیند <**iostream**> تعریف کننده شی‌های **cin**، **count**، **cerr** و **clog** است، که متناظر با استریم ورودی استاندارد و استریم خروجی استاندارد، خطای استریم استاندارد بافر نشده و خطای استریم استاندارد بافر شده می‌باشند. این فایل سرویس‌های برای هر دو نوع I/O قالب‌بندی شده و نشده عرضه می‌کند.

سرآیند <**iomanip**> سرویس‌های سودمندی برای I/O قالب‌بندی شده که دستکاری کننده‌های استریم پارامتری شده نامیده می‌شوند، همانند **setw** و **setprecision** فراهم می‌آورد.

سرآیند <**fstream**> سرویس‌های برای پردازش فایل کنترل شده توسط کاربر فراهم می‌آورد. در فصل ۱۷ از این سرآیند در برنامه‌ها استفاده خواهیم کرد.

۳-۲-۱۵ کلاس و شی‌های استریم ورودی/خروجی

کتابخانه **iostream** الگوهای متعددی برای کار با عملیات رایج I/O فراهم می‌آورد. برای مثال، الگوی کلاس **basic_istream** از عملیات استریم ورودی، الگوی کلاس **basic_ostream** از عملیات استریم

خروجی، و الگوی کلاس **basic_iostream** از هر دو استریم ورودی و خروجی پشتیبانی می‌کند. هر الگو دارای یک الگوی تخصصی از پیش تعریف شده است که امکان char I/O را فراهم می‌آورد.

علاوه بر این کتابخانه **iostream** مجموعه‌ای از *typedef*ها فراهم آورده است که اسامی مستعار برای این الگوهای تخصصی شده هستند. تصریح کننده **typedef** اسامی مستعار برای نوع داده‌های قبلاً تعریف شده فراهم می‌آورد. گاهی اوقات برنامه‌نویسان از **typedef** برای ایجاد اسامی کوتاهتر یا با معنی‌تر استفاده می‌کنند. برای مثال، عبارت

```
Typedef Card *CardPtr;
```

یک نوع اضافی بنام، **CardPtr** بعنوان یک نام مستعار برای نوع **Card *** تعریف می‌کند. توجه کنید که ایجاد یک نام با استفاده از **typedef** باعث ایجاد یک نوع داده نمی‌شود، **typedef** فقط یک نام برای نوعی که در برنامه بکار گرفته می‌شود، ایجاد می‌کند.

سلسله مراتب الگوی استریم I/O وسر بارگذاری عملگر

هر دو الگوی **basic_istream** و **basic_ostream** از طریق ارث‌بری مشترک از الگوی مبنا **basic_ios** کار می‌کنند. الگوی **basic_iostream** از طریق توارث مضاعف از الگوهای **basic_istream** و **basic_ostream** عمل می‌کند. دیاگرام کلاس در UML در شکل ۱-۱۵ آورده شده است، که روابط ارث‌بری را در میان آنها نشان می‌دهد.

سربارگذاری عملگر یک روش مناسب برای کار با I/O است. عملگر شیفت به چپ (<<) سربارگذاری شده مناسب استریم خروجی بوده از آن بعنوان عملگر درج استریم یاد می‌شود. عملگر شیفت به راست (>>) سربارگذاری شده مناسب برای استریم ورودی بوده و از آن بعنوان عملگر استخراج استریم یاد می‌شود. از این عملگرها به همراه شی‌های استریم استاندارد **cin**، **cout**، **cerr** و **clog** و گاهاً با شی‌های استریم تعریف شده از سوی کاربر استفاده می‌شود.

شکل ۱-۱۵ | بخشی از سلسله مراتب الگوی استریم I/O.

شی‌های استریم استاندارد cin، cout، cerr و clog

شی از قبل تعریف شده **cin** یک **istream** بوده و گفته می‌شود که "متصل به" (یا الصاق شده به) دستگاه ورودی استاندارد است که معمولاً صفحه کلید می‌باشد. عملگر استریم استخراج (>>)که در عبارت زیر بکار گرفته شده است سبب می‌شودتا مقدارمتغییر صحیح **grade** (با فرض اینکه **grade** قبلاً بعنوان یک متغیر **int** اعلان شده است) تا از طریق **cin** وارد حافظه گردد:

```
cin >> grade; //data "flows" in the direction of the arrows
```

توجه کنید که کامپایلر تعیین کننده نوع داده **grade** بوده و عملگر بارگذاری شده استریم استخراج را که مناسب است، انتخاب می‌کند. با فرض اینکه **grade** بدرستی اعلان شده باشد، عملگر استریم استخراج

مستلزم اطلاعات دیگری درباره نوع نمی‌باشد. عملگر << برای وارد کردن آیتم‌های داده از نوع توکار، رشته‌ها و اشاره‌گرها سربارگذاری شده است.

شی از پیش تعریف شده **cout** یک نمونه از **ostream** می‌باشد و گفته می‌شود که "متصل به" دستگاه خروجی استاندارد است که معمولاً صفحه نمایش است. عملگر >> که در عبارت زیر بکار گرفته شده است، سبب می‌شود تا مقدار متغیر **grade** از حافظه به دستگاه خروجی استاندارد منتقل شود:

```
cout << grade; //data "flows" in the direction of the arrows
```

همچنین توجه کنید که کامپایلر نوع داده **grade** را تعیین می‌کند (با فرض اینکه **grade** قبلاً بدرستی اعلان شده باشد) و عملگر >> انتخاب می‌شود و از اینرو این عملگر دیگر نیازی به اطلاعات اضافی در مورد نوع ندارد. عملگر >> باعث سربارگذاری آیتم‌های داده خروجی از نوع‌های توکار، رشته‌ها و مقادیر اشاره‌گر می‌شود.

شی از پیش تعریف شده **cerr** یک نمونه از **ostream** می‌باشد و گفته می‌شود که "متصل به" دستگاه خطای استاندارد می‌باشد. خروجی‌ها به شی **cerr** بافر نشده (*unbuffered*) هستند، و این مطلب را می‌رساند که هر گونه درج استریم **cerr** باعث می‌شود که خروجی آن بلافاصله ظاهر شود، این ویژگی روش مناسبی برای نشان دادن خطاها به کاربر است.

شی از پیش تعریف شده **clog** یک نمونه از کلاس **ostream** می‌باشد و گفته می‌شود که "متصل به" دستگاه خطای استاندارد است. خروجی‌ها به **clog** بافر شده (*buffered*) هستند. به این معنی که هر درجی به **clog** سبب می‌شود تا خروجی در یک بافر نگهداری شود تا زمانیکه بافر پر شود یا اینکه بافر خالی گردد. عملیات بافر کردن I/O از جمله تکنیک‌های است که در درس سیستم‌های عامل توضیح داده می‌شود.

الگوهای پردازش فایل

پردازش فایل در C++ با استفاده از الگوهای کلاس **basic_ifstream** (برای فایل ورودی)، **basic_ofstream** (برای فایل خروجی) و **basic_fstream** (برای فایل ورودی و خروجی) صورت می‌گیرد. هر الگوی کلاس دارای یک الگوی تخصصی شده از پیش تعریف شده است که امکان char I/O را فراهم می‌آورد. C++ مجموعه‌ای از typedefها تدارک است که اسامی مستعار برای این الگوهای تخصصی شده فراهم می‌آورد. برای مثال **typedef ifstream** نشاندهنده یک **basic_ifstream** تخصصی شده است که امکان می‌دهد **char** به یک فایل وارد گردد. همچنین **typedef fstream** نشاهنده یک **basic_fstream** است که امکان می‌دهد **char** از یک فایل خارج و به آن وارد گردد. الگوی **basic_ifstream** از **basic_iostream** ارث‌بری دارد. دیاگرام کلاس UML در شکل ۲-۱۵ آورده شده و روابط ارث‌بری در میان کلاس‌های مرتبط با I/O را نشان می‌دهد.

شکل ۲-۱۵ بخشی از سلسله مراتب الگوی استریم I/O در پردازش فایل.

۳-۱۵ استریم خروجی

خروجی قالب‌بندی شده و نشده توسط **ostream** تدارک دیده می‌شود. قابلیت‌های در نظر گرفته شده برای خروجی شامل انواع داده استاندارد با عملگر >>، چاپ کاراکترها از طریق تابع عضو **put**، خروجی قالب‌بندی نشده از طریق تابع عضو **write** (بخش ۵-۱۵)، چاپ مقادیر صحیح با فرمت‌های دسیمال (دهدهی)، اکتال و هگزادسیمال (بخش ۱-۶-۱۵)، چاپ مقادیر اعشاری با دقت‌های متفاوت (بخش ۲-۶-۱۵)، با توجه به نقطه اعشار (بخش ۱-۷-۱۵)، علامت‌گذاری علمی و ثابت (بخش ۵-۷-۱۵)، ترازبندی و تنظیم طول میدان داده (بخش ۲-۷-۱۵)، چاپ داده در لایه‌های مشخص شده (بخش ۷-۲-۱۵) و چاپ حروف بزرگ در علامت‌گذاری علمی و هگزادسیمال (بخش ۶-۷-۱۵) است.

۱-۳-۱۵ چاپ متغیرهای char *

C++ از قابلیت اتوماتیک تعیین نوع داده به نسبت C برخوردار است. متاسفانه گاهی اوقات این ویژگی به اشتباه می‌رود. برای مثال فرض کنید، می‌خواهیم مقدار یک **char *** را با یک رشته کاراکتری چاپ کنیم (یعنی آدرس حافظه اولین کاراکتر از رشته) با این وجود، عملگر >> برای چاپ داده از نوع **char *** بعنوان یک رشته خاتمه یافته با **null** سربارگذاری شده است. راه حل این مشکل تبدیل **char *** به **void *** است. برنامه شکل ۳-۱۵ مبادرت به چاپ یک متغیر **char *** در هر دو فرمت رشته و آدرس کرده است. توجه کنید که آدرس بصورت یک عدد هگزادسیمال (برمبنای 16) چاپ شده است. در بخش‌های ۱-۶-۱۵، ۴-۷-۱۵، ۵-۷-۱۵ و ۷-۷-۱۵ با نحوه کنترل مبنای اعداد آشنا خواهید شد.

```
1  // Fig. 15.3: Fig15_03.cpp
2  // Printing the address stored in a char * variable.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9     char *word = "again";
10
11    // display value of char *, then display value of char *
12    // static_cast to void *
13    cout << "Value of word is: " << word << endl
14       << "Value of static_cast< void * >( word ) is: "
15       << static_cast< void * >( word ) << endl;
16    return 0;
17 } // end main
```

```
Value of word is: again
Value of static_cast< void * >( word ) is: 00428300
```

شکل ۳-۱۵ | چاپ آدرس ذخیره شده در یک متغیر char *.

۲-۳-۱۵ چاپ کاراکتر با استفاده از تابع عضو put

می‌توانیم از تابع عضو **put** در چاپ کاراکترها استفاده کنیم. برای مثال، عبارت

```
cout.put( 'A' );
```

کاراکتر منفرد **A** را به نمایش در می‌آورد. فراخوانی تابع **put** می‌تواند بصورت پشت سرهم یا آبشاری باشد، همانند عبارت

```
cout.put( 'A' ).put( '\n' );
```

که حرف **A** و بدنبال آن یک کاراکتر خط جدید چاپ می‌شود. همانند عملگر >> در عبارت قبلی، چون عملگر نقطه (.) از سمت چپ به راست ارزیابی می‌شود، و تابع **put** یک مراجعه به شی **ostream** (یعنی **cout**) برگشت می‌دهد که فراخوانی **put** را دریافت می‌کند. همچنین امکان دارد تابع **put** با یک عبارت عددی که نشاندهنده یک مقدار ASCII است فراخوانی شود، همانند عبارت زیر

```
cout.put( 65 );
```

که **A** را چاپ می‌کند.

۱۵-٤ استریم ورودی

حال اجازه دهید تا نگاهی به استریم ورودی داشته باشیم. قابلیت ورودی قالب‌بندی شده و نشده توسط **istream** تدارک دیده شده است. عملگر >> معمولاً کاراکترهای *white-space* (همانند فضاهای خالی، تب‌ها و خطوط جدید) را در استریم ورودی در نظر نمی‌گیرد. همانطوری که بعداً خواهید دید می‌توانیم این رفتار را تغییر دهیم. پس از هر ورودی، عملگر >> یک مراجعه به شی استریم که پیغام استخراج را دریافت کرده است، برگشت می‌دهد (مثلاً، **cin** در عبارت **grade**>>**cin**). اگر آن مراجعه در یک شرط بکار گرفته شود (مثلاً در شرط تکرار حلقه **while**)، عملگر سربارگذاری شده **void *** بطور ضمنی احضار شده و مراجعه را تبدیل به یک مقدار اشاره‌گر غیر **null** یا اشاره‌گر **null** براساس موفقیت یا عدم موفقیت، آخرین عملیات ورودی می‌کند. اشاره‌گر غیر **null** به **true** تبدیل می‌شود تا نشاندهنده موفقیت‌آمیز بودن باشد و اشاره‌گر **null** به مقدار **false** تبدیل می‌شود تا دلالت بر عدم موفقیت کند. زمانیکه مبادرت به خواندن انتهای استریم می‌شود، عملگر تبدیل **void *** یک اشاره‌گر **null** برگشت می‌دهد تا نشاندهنده انتهای فایل باشد.

هر شی استریم حاوی مجموعه‌ای از بیت‌های وضعیت است که در کنترل وضعیت استریم نقش دارند (یعنی، قالب‌بندی، تنظیم خطا و غیره). این بیت‌ها توسط عملگر تبدیل **void *** برای تعیین اینکه آیا یک اشاره‌گر غیر **null** یا اشاره‌گر **null** برگشت داده شده است، استفاده می‌شود. اگر داده وارد شده از نوع اشتباه باشد، **failbit** تنظیم می‌شود و اگر عملیات با شکست مواجه شود، **badbit** تنظیم می‌گردد. در بخش‌های ۱۵-۷ و ۱۵-۸ به بررسی این بیت‌ها خواهیم پرداخت.

۱۵-٤-۱ توابع عضو get و getline

تابع عضو **get** بدون آرگومان، سبب می‌شود تا یک کاراکتر از استریم معین شده وارد گردد (شامل کاراکترهای *white-space* و سایر کاراکترهای غیرگرافیکی) و آنرا بعنوان مقداری از فراخوانی تابع برگشت می‌دهد. این نسخه از **get** زمانیکه به انتهای فایل در استریم برسد، **EOF** برگشت می‌دهد.

استفاده از توابع عضو eof، get و put

برنامه شکل ۴-۱۵ به توصیف نحوه استفاده از توابع عضو **eof** و **get** بر روی استریم ورودی **cin** و تابع عضو **put** بر روی استریم خروجی **cout** می‌پردازد. برنامه ابتدا مقدار **cin.eof()** را چاپ می‌کند. یعنی **false** (0 در خروجی)، تا نشان دهد که انتهای فایل در **cin** رخ نداده است. کاربر یک عبارت متنی وارد کرده و کلید *Enter* را بدنبال انتهای فایل فشار می‌دهد (*<ctrl>-z*) بر روی سیستم ویندوز مایکروسافت، *<ctrl>-d*) بر روی سیستم‌های لینوکس و مکنتاش). خط 17 هر کاراکتر را می‌خواند و خط 18 با استفاده از تابع عضو **put** آنرا در **cout** قرار می‌دهد. زمانیکه با انتهای فایل مواجه شود، عبارت **while** خاتمه یافته و خط 22 مقدار **cin.eof()** را که اکنون **true** است (1 در خروجی) چاپ می‌کند تا نشان دهد که انتهای فایل با **cin** تنظیم شده است. دقت کنید که در این برنامه از نسخه **get** متعلق به **istream** استفاده شده است که آرگومانی دریافت نمی‌کند و کاراکتر ورودی را برگشت می‌دهد (خط 17). تابع **eof** فقط پس از رسیدن برنامه یا اقدام به خواندن آخرین کاراکتر پس از استریم، **true** برگشت می‌دهد.

```
1  // Fig. 15.4: Fig15_04.cpp
2  // Using member functions get, put and eof.
3  #include <iostream>
4  using std::cin;
5  using std::cout;
6  using std::endl;
7
8  int main()
9  {
10    int character; // use int, because char cannot represent EOF
11
12    // prompt user to enter line of text
13    cout << "Before input, cin.eof() is " << cin.eof() << endl
14       << "Enter a sentence followed by end-of-file:" << endl;
15
16    // use get to read each character; use put to display it
17    while ( ( character = cin.get() ) != EOF )
18       cout.put( character );
19
20    // display end-of-file character
21    cout << "\nEOF in this system is: " << character << endl;
22    cout << "After input of EOF, cin.eof() is " << cin.eof() << endl;
23    return 0;
24 } // end main
```

```
Before input, cin.eof() is 0
Enter a sentence followed by end-of-file:
Testing the get and put member functions
Testing the get and put member functions
^z

EOF in this system is: -1
After input of EOF, cin.eof() is 1
```

شکل ٤-١٥ | توابع عضو get، put و eof.

تابع عضو **get** با یک آرگومان مراجعه–کاراکتر مبادرت به وارد کردن کاراکتر بعدی از استریم ورودی می‌کند (حتی اگر یک کاراکتر *white-space* باشد) و آن را در آرگومان کاراکتری ذخیره می‌سازد. این نسخه از **get** یک مراجعه به شی **istream** برگشت می‌دهد.

سومین نسخه از تابع **get** سه آرگومان دریافت می‌کند، یک آرایه کاراکتری، حد سایز و یک حائل (با مقدار پیش‌فرض **'\n'**). این نسخه کاراکترها را از استریم ورودی می‌خواند. تابع به تعداد یکی کمتر از حداکثر تعداد کاراکترهای تعیین شده را خوانده و یا بلافاصله پس از خواندن حائل بکار خاتمه می‌دهد. از یک کاراکتر **null** وارد شده به رشته ورودی در یک آرایه کاراکتری بعنوان بافر در برنامه استفاده می‌شود. حائل در آرایه کاراکتری جای نمی‌گیرد اما در استریم ورودی باقی می‌ماند. از اینرو، نتیجه دومین فراخوانی پی‌درپی **get** یک خط خالی است، مگر اینکه کاراکتر حائل از استریم ورودی حذف گردد (اینکار با **cin.ignore()** امکان‌پذیر است).

مقایسه cin و cin.get

برنامه شکل ۵-۱۵ به مقایسه ورودی با استفاده از **cin** و **cin.get** پرداخته است. دقت کنید که در فراخوانی **cin.get** (خط 24) حائلی مشخص نشده است، از اینرو کاراکتر پیش‌فرض **'\n'** بکار گرفته شده است.

```
1  // Fig. 15.5: Fig15_05.cpp
2  // Contrasting input of a string via cin and cin.get.
3  #include <iostream>
4  using std::cin;
5  using std::cout;
6  using std::endl;
7
8  int main()
9  {
10    // create two char arrays, each with 80 elements
11    const int SIZE = 80;
12    char buffer1[ SIZE ];
13    char buffer2[ SIZE ];
14
15    // use cin to input characters into buffer1
16    cout << "Enter a sentence:" << endl;
17    cin >> buffer1;
18
19    // display buffer1 contents
20    cout << "\nThe string read with cin was:" << endl
21       << buffer1 << endl << endl;
22
23    // use cin.get to input characters into buffer2
24    cin.get( buffer2, SIZE );
25
26    // display buffer2 contents
27    cout << "The string read with cin.get was:" << endl
28       << buffer2 << endl;
29    return 0;
30 } // end main
```

```
Enter a sentence:
Contrasting string input with cin and cin.get

The string read with cin was:
Contrasting

The string read with cin.get was:
 string input with cin and cin.get
```

شکل ۵-۱۵ | وارد کردن رشته با استفاده از cin در مقایسه با cin.get

استفاده از تابع عضو getline

عملکرد تابع عضو **getline** شبیه سومین نسخه از تابع عضو **get** است و یک کاراکتر **null** پس از خط در آرایه کاراکتری وارد می‌کند. تابع **getline** مبادرت به حذف حائل از استریم می‌کند (یعنی کاراکتر را خواند و دور می‌اندازد)، اما آنرا در آرایه کاراکتری ذخیره نمی‌کند. برنامه شکل ۶-۱۵ به بررسی استفاده از تابع **getline** در وارد کردن یک خط متنی پرداخته است.

```
1  // Fig. 15.6: Fig15_06.cpp
2  // Inputting characters using cin member function getline.
3  #include <iostream>
4  using std::cin;
5  using std::cout;
6  using std::endl;
7
8  int main()
9  {
10    const int SIZE = 80;
11    char buffer[ SIZE ]; // create array of 80 characters
12
13    // input characters in buffer via cin function getline
14    cout << "Enter a sentence:" << endl;
15    cin.getline( buffer, SIZE );
16
17    // display buffer contents
18    cout << "\nThe sentence entered is:" << endl << buffer << endl;
19    return 0;
20 } // end main
```

```
Enter a sentence:
Using the getline member function

The sentence entered is:
Using the getline member function
```

شکل ٦-۱۵ وارد کردن داده کاراکتری با cin تابع عضو getline.

۱۵-٤-۲ توابع عضو peek، putback و ignore

تابع عضو **ignore** از **istream** قادر به خواندن و دور انداختن کاراکترها به تعداد مشخص شده (مقدار پیش‌فرض یک کاراکتر است) یا خاتمه دادن به خواندن در مواجه شدن با حائل (حائل پیش‌فرض **EOF** می‌باشد که سبب می‌شود **ignore** به انتهای فایل پرش کند) می‌گردد.

تابع عضو **putback** کاراکتر قبلی بدست آمده از **get** از استریم ورودی را به استریم باز می‌گرداند. این تابع مناسب برای برنامه‌های است که یک استریم ورودی را اسکن می‌کنند تا به انتهای یک فیلد با کاراکتر مشخص برسند. زمانیکه این کاراکتر وارد شد، برنامه کاراکتر را به استریم باز می‌گرداند، از اینرو کاراکتر می‌تواند در داده ورودی لحاظ شود.

تابع عضو **peek** کاراکتر بعدی از یک استریم ورودی را برگشت می‌دهد، اما کاراکتر را از استریم حذف نمی‌کند.

۱۵-٤-۳ I/O نوع ایمن

زبان C++ ارائه کننده I/O از نوع ایمن است. عملگرهای >> و << سربارگذاری شده تا آیتم‌های داده از نوع مشخص را پذیرا باشند. اگر داده غیرمنتظره‌ای پردازش شود، بیت‌های مختلفی تنظیم می‌شوند که کاربر می‌تواند با بررسی آنها به وضعیت عملیات I/O پی ببرد.

۱۵-۵ ورودی/خروجی قالب‌بندی نشده با استفاده از read، write و gcout

ورودی/خروجی قالب‌بندی نشده با استفاده از توابع عضو **read** و **write** از **istream** و **ostream** صورت می‌گیرد. تابع عضو **read** تعدادی از بایت‌ها را به آرایه کاراکتری در حافظه وارد می‌سازد، تابع عضو **write** بایت‌های از آرایه کاراکتری خارج می‌کند.

این بایت‌ها قالب‌بندی شده نیستند. آنها بصورت بایت‌های خام وارد یا خارج می‌شوند. برای مثال در فراخوانی

```
char buffer[] = "HAPPY BIRTHDAY";
cout.write( buffer, 10 );
```

ده بایت اول از **buffer** چاپ می‌شود. فراخوانی

```
cout.write( "ABCDEFGHIJKLMNOPQRSTUVWZYZ", 10 );
```

ده کاراکتر اول از الفبا را نشان می‌دهد.

تابع عضو **read** به تعداد مشخص شده‌ای از کاراکترها را به یک آرایه کاراکتری می‌خواند. اگر کمتر از تعداد مشخص شده، کاراکتر قرائت شود، **failbit** تنظیم خواهد شد. در بخش ۸-۱۵ با نحوه تعیین و تنظیم وضعیت **failbiy** آشنا خواهید شد. تابع عضو **gcount** گزارشی از کاراکترهای خوانده شده توسط آخرین عملیات ورودی تهیه می‌کند.

برنامه شکل ۷-۱۵ به بررسی عملکرد توابع عضو **read** و **gcount** از **isream** و تابع عضو **write** از **ostream** پرداخته است. برنامه **20** کاراکتر دریافت و در آرایه کاراکتری **buffer** با دستور **read** قرار می‌دهد (خط 15)، تعداد کاراکترهای ورودی توسط **gcount** تعیین می‌شود (خط 19) و کاراکترهای موجود در **burffer** توسط دستور **write** چاپ می‌شوند. (خط 19).

```
1  // Fig. 15.7: Fig15_07.cpp
2  // Unformatted I/O using read, gcount and write.
3  #include <iostream>
4  using std::cin;
5  using std::cout;
6  using std::endl;
7
8  int main()
9  {
10    const int SIZE = 80;
11    char buffer[ SIZE ]; // create array of 80 characters
12
13    // use function read to input characters into buffer
14    cout << "Enter a sentence:" << endl;
15    cin.read( buffer, 20 );
16
17    // use functions write and gcount to display buffer characters
18    cout << endl << "The sentence entered was:" << endl;
19    cout.write( buffer, cin.gcount() );
20    cout << endl;
```

```
21    return 0;
22 } // end main
```

```
Enter a sentence:
Using the read, write, and gcount member functions
The sentence enterd was:
Using the read, write
```

شکل ۷-۱۵ I/O قالب‌بندی نشده با استفاده از توابع read، gcount و wtire.

۶-۱۵ معرفی دستکاری کننده‌های استریم

زبان C++ تعداد متنابهی دستکاری کننده استریم در نظر گرفته است که وظایف قالب‌بندی را برعهده دارند. این وظایف عبارتند از تنظیم طول میدان، تنظیمات مربوط به دقت، تنظیم وضعیت قالب‌بندی، تنظیم کاراکتر پرکننده در میدان، دور ریختن استریم‌ها، وارد کردن خط جدید به استریم ورودی (و حذف خطی از استریم)، وارد کردن کاراکتر null به استریم خروجی و در نظر نگرفتن کاراکترهای *white-space* در استریم ورودی. این ویژگیهای در بخش‌های زیر توضیح داده می‌شوند.

۱-۶-۱۵ پایه انتگرال استریم: dec، oct، hex و setbase

معمولاً اعداد صحیح بعنوان مقادیر دسیمال (پایه **10**) تفسیر می‌شوند. برای تغییر پایه و تفسیر مقادیر موجود در استریم، دستکاری کننده **hex** را برای تنظیم پایه به هگزادسیمال (پایه **16**) یا وارد کردن **oct** برای تنظیم پایه به اکتال (پایه **8**) بکار می‌گیرند. با وارد کردن دستکاری کننده **dec** به استریم پایه به حالت دسیمال باز می‌گردد.

همچنین می‌توان با استفاده از دستکاری کننده استریم **setbase** مبادرت به تغییر پایه کرده که یک آرگومان از 10، 8 یا 16 دریافت می‌کند تا به ترتیب تنظیم کننده پایه به دسیمال، اکتال یا هگزادسیمال باشد. استفاده از **setbase** (یا هر نوع دستکاری کننده پارامتری) مستلزم استفاده از فایل سرآیند <**iomanip**> است. مقادیر پایه استریم تا زمانیکه تغییری در آن داده نشود، وضعیت خود را حفظ می‌کند.

برنامه شکل ۸-۱۵ به بررسی **hex**، **oct**، **dec** و **setbase** پرداخته است.

```
1  // Fig. 15.8: Fig15_08.cpp
2  // Using stream manipulators hex, oct, dec and setbase.
3  #include <iostream>
4  using std::cin;
5  using std::cout;
6  using std::dec;
7  using std::endl;
8  using std::hex;
9  using std::oct;
10
11 #include <iomanip>
12 using std::setbase;
13
14 int main()
15 {
16    int number;
17
18    cout << "Enter a decimal number: ";
19    cin >> number; // input number
20
21    // use hex stream manipulator to show hexadecimal number
22    cout << number << " in hexadecimal is: " << hex
23       << number << endl;
```

```
24
25    // use oct stream manipulator to show octal number
26    cout << dec << number << " in octal is: "
27       << oct << number << endl;
28
29    // use setbase stream manipulator to show decimal number
30    cout << setbase( 10 ) << number << " in decimal is: "
31       << number << endl;
32    return 0;
33 } // end main
```

```
Enter a decimal number: 20
20 in hexadecimal is: 14
20 in octal is: 24
20 in decimal is: 20
```

شکل ۸-۱۵ دستکاری کننده‌های استریم hex، oct، dec و setbase.

۲-٦-۱۵ دقت نقطه اعشار (setprecision,precision)

می‌توانیم با استفاده از دستکاری کننده **setprecision** یا تابع عضو **precision** از **io_base** مبادرت به تنظیم دقت اعداد اعشاری کنیم (یعنی تعداد ارقام در سمت راست نقطه اعشار). این تنظیم صورت گرفته بر روی تمام خروجی‌ها اعمال می‌شود تا اینکه تنظیم دیگری در دقت اعمال گردد. فراخوانی تابع عضو **precision** بدون آرگومان، دقت جاری را برگشت می‌دهد. برنامه شکل ۹-۱۵ از تابع عضو **precision** در خط 28 و دستکاری کننده **setprecsion** در خط 37 برای چاپ یک جدول که نشاندهنده ریشه دوم عدد 2 با دقت اعمال شده از 0-9 استفاده کرده است.

```
1  // Fig. 15.9: Fig15_09.cpp
2  // Controlling precision of floating-point values.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6  using std::fixed;
7
8  #include <iomanip>
9  using std::setprecision;
10
11 #include <cmath>
12 using std::sqrt; // sqrt prototype
13
14 int main()
15 {
16    double root2 = sqrt( 2.0 ); // calculate square root of 2
17    int places; // precision, vary from 0-9
18
19    cout << "Square root of 2 with precisions 0-9." << endl
20       << "Precision set by ios_base member function "
21       << "precision:" << endl;
22
23    cout << fixed; // use fixed point format
24
25    // display square root using ios_base function precision
26    for ( places = 0; places <= 9; places++ )
27    {
28       cout.precision( places );
29       cout << root2 << endl;
30    } // end for
31
32    cout << "\nPrecision set by stream manipulator "
33       << "setprecision:" << endl;
34
35    // set precision for each digit, then display square root
36    for ( places = 0; places <= 9; places++ )
37       cout << setprecision( places ) << root2 << endl;
```

```
38
39    return 0;
40 } // end main
```

```
Square root of 2 with precisions 0-9.
Precision set by iso_base member function precision:
1
1.4
1.41
1.414
1.4142
1.41421
1.414214
1.4142136
1.41421356
1.414213562

Precision set by stream manipulator setprecision:
1
1.4
1.41
1.414
1.4142
1.41421
1.414214
1.4142136
1.41421356
1.414213562
```

شکل ۹-۱۵ | دقت مقادیر اعشاری.

۳-۶-۱۵ طول میدان (width و setw)

تابع عضو **width** (از کلاس مبنای **iso_base**) مبادرت به تنظیم طول میدان می‌کند و طول میدان قبلی را برگشت می‌دهد. اگر مقادیر خروجی کمتر از طول میدان باشند، کاراکترهای پرکننده بعنوان لایه (padding) بکار گرفته می‌شوند. مقدار بزرگتر از پهنای در نظر گرفته شده قطع نمی‌شود، کل عدد چاپ می‌شود. تابع **width** بدون آرگومان، تنظیم جاری را برگشت می‌دهد.

در برنامه شکل ۱۰-۱۵ به بررسی نحوه استفاده از تابع عضو **width** هم بر روی ورودی و هم خروجی پرداخته شده است. همچنین از دستکاری کننده **setw** برای تنظیم طول میدان استفاده می‌شود.

```
1  // Fig. 15.10: Fig15_10.cpp
2  // Demonstrating member function width.
3  #include <iostream>
4  using std::cin;
5  using std::cout;
6  using std::endl;
7
8  int main()
9  {
10    int widthValue = 4;
11    char sentence[ 10 ];
12
13    cout << "Enter a sentence:" << endl;
14    cin.width( 5 ); // input only 5 characters from sentence
15
16    // set field width, then display characters based on that width
17    while ( cin >> sentence )
18    {
19       cout.width( widthValue++ );
20       cout << sentence << endl;
21       cin.width( 5 ); // input 5 more characters from sentence
22    } // end while
23
24    return 0;
```

```
25 } // end main
```

```
Enter a sentence:
This is a test of the width member function
This
   is
     a
   test
      of
      the
      widt
          h
        memb
           er
          func
           tion
```

شکل ۱۰-۱۵ تابع عضو width از کلاس ios_base.

۱۵-٦-٤ دستکاری کننده‌های استریم خروجی تعریف شده توسط کاربر

برنامه‌نویسان می‌تواند دستکاری کننده‌های استریم متعلق بخود را ایجاد کنند. در برنامه شکل ۱۱-۱۵ ایجاد و استفاده از دستکاری کننده‌های استریم غیر پارامتری و جدید **bell** (خطوط 13-10)، **carriageReturn** (خطوط 19-16)، **tab** (خطوط 25-22) و **endLine** (خطوط 32-29) نشان داده شده است. برای دستکاری کننده‌های استریم خروجی، نوع برگشتی و پارامتر بایستی از نوع **ostream &** باشد. زمانیکه خط 37 مبادرت به وارد کردن **endLine** در استریم خروجی می‌کند، تابع **endLine** فراخوانی شده و خط 31 دنباله **n** را و دستکاری کننده **flush** را بر روی استریم استاندارد **cout** خارج می‌سازد. به همین ترتیب، زمانیکه خطوط 46-37 دستکاری کننده‌های **tab**، **bell** و **carrigeReturn** را وارد استریم خروجی می‌سازند، توابع متناظر **tab** (خط 22)، **bell** (خط 10) و **carriageReturn** (خط 16) فراخوانی می‌شوند.

```
1  // Fig. 15.11: Fig15_11.cpp
2  // Creating and testing user-defined, nonparameterized
3  // stream manipulators.
4  #include <iostream>
5  using std::ostream;
6  using std::cout;
7  using std::flush;
8
9  // bell manipulator (using escape sequence \a)
10 ostream& bell( ostream& output )
11 {
12    return output << '\a';  // issue system beep
13 } // end bell manipulator
14
15 // carriageReturn manipulator (using escape sequence \r)
16 ostream& carriageReturn( ostream& output )
17 {
18    return output << '\r';  // issue carriage return
19 } // end carriageReturn manipulator
20
21 // tab manipulator (using escape sequence \t)
22 ostream& tab( ostream& output )
23 {
24    return output << '\t';  // issue tab
25 } // end tab manipulator
26
27 // endLine manipulator (using escape sequence \n and member
28 // function flush)
29 ostream& endLine( ostream& output )
30 {
31    return output << '\n' << flush; // issue endl-like end of line
```

```
32 } // end endLine manipulator
33
34 int main()
35 {
36    // use tab and endLine manipulators
37    cout << "Testing the tab manipulator:" << endLine
38       << 'a' << tab << 'b' << tab << 'c' << endLine;
39
40    cout << "Testing the carriageReturn and bell manipulators:"
41       << endLine << ".........";
42
43    cout << bell; // use bell manipulator
44
45    // use ret and endLine manipulators
46    cout << carriageReturn << "-----" << endLine;
47    return 0;
48 } // end main
```

```
Testing the tab manipulator:
a       b       c
Testing the carriageReturn and bell manipulators:
-----.....
```

شکل ۱۱-۱۵ | دستکاری کننده‌های استریم غیر پارامتری تعریف شده توسط کاربر.

۱۵-۷ تعیین فرمت استریم و دستکاری کننده‌های استریم

می‌توان از انواع دستکاری کننده‌های استریم به منظور تصریح نوع قالب‌بندی در حین عملیات I/O استفاده کرد. دستکاری کننده‌های استریم بر قالب‌بندی خروجی کنترل دارند. در جدول شکل ۱۲-۱۵ دستکاری کننده‌های استریم که بر قالب استریم کنترل دارند، لیست شده‌اند. تمام این دستکاری کننده‌ها متعلق به کلاس **ios_base** می‌باشند. در بخش‌های بعدی از این دستکاری کننده‌ها در مثال‌ها استفاده کرده‌ایم.

دستکاری کننده	توضیح
skipws	از کاراکترهای *white-space* در استریم ورودی صرفنظر (پرش) می‌کند. اینحالت با استفاده از **noskipws** از کار می‌افتد.
left	خروجی را از سمت چپ تنظیم (تراز) می‌کند. کاراکترهای لایه‌گذاری در صورت نیاز در سمت راست ظاهر می‌شوند.
right	خروجی را از سمت راست تنظیم (تراز) می‌کند. کاراکترهای پایه لایه‌گذاری در صورت نیاز در سمت چپ ظاهر می‌شوند.
internal	نشان می‌دهد که علامت عدد باید از سمت چپ و بزرگی عدد باید از سمت راست تنظیم شود.
dec	مشخص می‌کند که با مقادیر صحیح همانند مقادیر دسیمال (پایه **10**) رفتار شود.
oct	مشخص می‌کند که با مقادیر صحیح همانند مقادیر اکتال (پایه **8**) رفتار شود.
hex	مشخص می‌کند که با مقادر صحیح همانند مقادیر هگزادسیمال (پایه **16**) رفتار شود.
showbase	مشخص می‌کند که پایه یک عدد در کنار آن در خروجی قرار گیرد (**0** برای اکتال، **0x** برای هگزادسیمال). این تنظیم با استفاده از **noshowbase** از کار می‌افتد.
showpoint	مشخص می‌کند که اعداد اعشاری بایستی با نقطه دسیمال چاپ شوند. معمولاً این دستکاری کننده با **fixed** بکار گرفته می‌شود تا تضمینی برای تعداد ارقام مشخص شده در سمت راست نقطه دسیمال باشد، حتی اگر آنها صفر باشند.

uppercase	مشخص می‌کند که باید ازحروف بزرگ (یعنی **X** و **A** تا **F**) در یک عدد صحیح هگزادسیمال و حروف بزرگ **E** در نمایش یک مقدار اعشاری در نماد علمی بکار گرفته شود.
showpos	مشخص می‌کند که اعداد مثبت بایستی همراه با نماد جمع (+) ظاهر شوند.
scientific	خروجی با نمایش علمی ظاهر می‌گردد.
fixed	تعیین کننده نقطه ثابت در یک مقدار اعشاری به تعداد ارقام مشخص شده در سمت راست نقطه دسیمال است.

جدول ۱۲-۱۵ | دستکاری کننده‌های وضعیت قالب‌بندی از <iosream>.

۱-۷-۱۵ دنباله صفرها و نقاط دسیمال (showpoint)

دستکاری کننده **showpoint** یک عدد اعشاری را مجبور می‌کند تا در خروجی با نقطه دسیمال و دنباله‌ای از صفرهای تعیین شده ظاهر شود. برای مثال، مقدار اعشاری 79.0 بدون استفاده از **showpoint** بصورت 79 و به هنگام استفاده از **showpoint** بصورت 79.000000 (یا تعداد بیشتری از صفرهای دنباله که با دقت جاری مشخص می‌شوند) ظاهر می‌شود. نقطه مقابل **showpoint** دستکاری کننده **noshowpoint** است. برنامه موجود در شکل ۱۳-۱۵ نحوه استفاده از **showpoint** در کنترل چاپ دنباله‌ای از صفرها و نقاط دسیمال در مقادیر اعشاری را نشان می‌دهد. بخاطر داشته باشید که دقت پیش‌فرض برای یک عدد اعشاری، 6 است. زمانیکه از **fixed** یا **scientific** استفاده نشده باشد، دقت نمایش، تعداد ارقام بامعنی است، و نه تعداد ارقام پس از نقطه دسیمال.

```
1  // Fig. 15.13: Fig15_13.cpp
2  // Using showpoint to control the printing of
3  // trailing zeros and decimal points for doubles.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7  using std::showpoint;
8
9  int main()
10 {
11    // display double values with default stream format
12    cout << "Before using showpoint" << endl
13       << "9.9900 prints as: " << 9.9900 << endl
14       << "9.9000 prints as: " << 9.9000 << endl
15       << "9.0000 prints as: " << 9.0000 << endl << endl;
16
17    // display double value after showpoint
18    cout << showpoint
19       << "After using showpoint" << endl
20       << "9.9900 prints as: " << 9.9900 << endl
21       << "9.9000 prints as: " << 9.9000 << endl
22       << "9.0000 prints as: " << 9.0000 << endl;
23    return 0;
24 } // end main
```

```
Before using showpoint
9.9900 prints as: 9.99
9.9000 prints as: 9.9
9.0000 prints as: 9

After using showpoint
9.9900 prints as: 9.99000
9.9000 prints as: 9.90000
9.0000 prints as: 9.00000
```

شکل ۱۳-۱۵ | کنترل چاپ دنباله‌ای از صفرها و نقاط دسیمال در مقادیر اعشاری.

۱۵-۷-۲ ترازبندی (left، right و internal)

دستکاری کننده‌های **left** و **right** به فیلدها امکان می‌دهند تا به ترتیب از سمت چپ با کاراکترهای لایه‌گذاری به سمت راست و از سمت راست با کاراکترهای لایه‌گذاری به سمت چپ تراز شوند. کاراکتر لایه‌گذاری توسط تابع عضو **fill** یا دستکاری کننده استریم پارامتری **setfill** مشخص می‌شود. در برنامه شکل ۱۴-۱۵ از دستکاری کننده‌های **setw**، **left** و **right** برای ترازبندی چپ و راست داده صحیح در یک فیلد استفاده شده است.

```
1  // Fig. 15.14: Fig15_14.cpp
2  // Demonstrating left justification and right justification.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6  using std::left;
7  using std::right;
8
9  #include <iomanip>
10 using std::setw;
11
12 int main()
13 {
14    int x = 12345;
15
16    // display x right justified (default)
17    cout << "Default is right justified:" << endl
18       << setw( 10 ) << x;
19
20    // use left manipulator to display x left justified
21    cout << "\n\nUse std::left to left justify x:\n"
22       << left << setw( 10 ) << x;
23
24    // use right manipulator to display x right justified
25    cout << "\n\nUse std::right to right justify x:\n"
26       << right << setw( 10 ) << x << endl;
27    return 0;
28 } // end main
```

```
Default is right justified:
     12345

Use std::left to left justify x:
12345

Use std::right to right justify x:
     12345
```

شکل ۱۴-۱۵ | ترازبندی چپ و راست با left و right.

دستکاری کننده استریم **internal** بر این نکته دلالت دارد که علامت عدد (یا پایه به هنگام استفاده از **showbase**) باید در سمت چپ فیلد تراز شود، و خود عدد از سمت راست تراز گردد و فاصله مناسب با لایه‌گذاری کاراکتر پرکننده حفظ گردد. برنامه شکل ۱۵-۱۵ نحوه استفاده از دستکاری کننده استریم **internal** با فاصله‌گذاری مشخص (خط 15) را نشان می‌دهد. توجه کنید که **showpos** نماد جمع را چاپ می‌کند (خط 15). برای غیرفعال کردن **showpos** می‌توان از **noshowpos** استفاده کرد.

```
1  // Fig. 15.15: Fig15_15.cpp
2  // Printing an integer with internal spacing and plus sign.
3  #include <iostream>
```

```
4  using std::cout;
5  using std::endl;
6  using std::internal;
7  using std::showpos;
8
9  #include <iomanip>
10 using std::setw;
11
12 int main()
13 {
14    // display value with internal spacing and plus sign
15    cout << internal << showpos << setw( 10 ) << 123 << endl;
16    return 0;
17 } // end main
+         123
```

شکل ۱۵-۱۵ | چاپ یک عدد صحیح با فاصله‌گذاری داخلی و نماد جمع.

۱۵-۷-۳ لایه‌گذاری (fill، setfill)

تابع عضو **fill** تعیین کننده کاراکتر پرکننده در ترازبندی فیلدها است، اگر هیچ مقداری مشخص نشود، برای لایه‌گذاری از فاصله‌ها استفاده خواهد شد. تابع **fill** کاراکتر لایه‌گذاری قبلی را برگشت می‌دهد. همچنین دستکاری کننده **setfill** مبادرت به تنظیم کاراکتر لایه‌گذاری می‌کند. در برنامه شکل ۱۶-۱۵ به بررسی نحوه استفاده از تابع عضو **fill** در خط 40 و دستکاری کننده استریم **setfill** در خطوط 47 و 44 پرداخته است.

```
1  // Fig. 15.16: Fig15_16.cpp
2  // Using member-function fill and stream-manipulator setfill to change
3  // the padding character for fields larger than the printed value.
4  #include <iostream>
5  using std::cout;
6  using std::dec;
7  using std::endl;
8  using std::hex;
9  using std::internal;
10 using std::left;
11 using std::right;
12 using std::showbase;
13
14 #include <iomanip>
15 using std::setfill;
16 using std::setw;
17
18 int main()
19 {
20    int x = 10000;
21
22    // display x
23    cout << x << " printed as int right and left justified\n"
24       << "and as hex with internal justification.\n"
25       << "Using the default pad character (space):" << endl;
26
27    // display x with base
28    cout << showbase << setw( 10 ) << x << endl;
29
30    // display x with left justification
31    cout << left << setw( 10 ) << x << endl;
32
33    // display x as hex with internal justification
34    cout << internal << setw( 10 ) << hex << x << endl << endl;
35
36    cout << "Using various padding characters:" << endl;
37
38    // display x using padded characters (right justification)
39    cout << right;
```

```
40    cout.fill( '*' );
41    cout << setw( 10 ) << dec << x << endl;
42
43    // display x using padded characters (left justification)
44    cout << left << setw( 10 ) << setfill( '%' ) << x << endl;
45
46    // display x using padded characters (internal justification)
47    cout << internal << setw( 10 ) << setfill( '^' ) << hex
48       << x << endl;
49    return 0;
50 } // end main
```

```
10000 printed as int right and left justified
and as hex with internal justification.
Using the default pad character (space):
     10000
10000
0x    2710

Using various padding characters:
*****10000
10000%%%%%
0x^^^^2710
```

شکل ١٦-١٥ | استفاده از تابع fill و دستکاری کننده setfill.

١٥-٧-٤ پایه انتگرال استریم (dec، oct، hex، showbase)

زبان C++ دستکاری کننده‌های استریم **dec**، **hex** و **oct** را برای تصریح اینکه مقادیر صحیح بصورت دسیمال، هگزادسیمال و اکتال به نمایش درآیند، در نظر گرفته است. اگر از هیچ یک از این دستکاری کننده‌ها استفاده نشود، حالت دسیمال بکار گرفته خواهد شد. از زمانیکه یک پایه خاص برای استریم تعیین شد، تمام مقادیر صحیح در آن استریم با آن پایه پردازش خواهند شد تا اینکه این پایه تغییر داده شود. در خروجی استریم با مقادیر صحیح با پسوند صفر همانند مقادیر اکتال، مقادیر پسوند صحیح با **0x** یا **0X** همانند مقادیر هگزادسیمال و تمام مقادیر صحیح همانند مقادیر دسیمال رفتار خواهد شد.

دستکاری کننده استریم **showbase** سبب می‌شود تا مبنای یک مقدار در خروجی ظاهر شود. اعداد دسیمال بطور عادی در خروجی دیده می‌شوند، اعداد اکتال با یک دنباله صفر، و اعداد هگزادسیمال با دنباله‌ای از **0x** یا **0X**. در برنامه شکل ١٧-١٥ به بررسی نحوه استفاده از **showbase** پرداخته شده است. برای از کار انداختن **showbase** می‌توانید از **noshowbase** استفاده کنید.

```
1  // Fig. 15.17: Fig15_17.cpp
2  // Using stream-manipulator showbase.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6  using std::hex;
7  using std::oct;
8  using std::showbase;
9
10 int main()
11 {
12    int x = 100;
13
14    // use showbase to show number base
15    cout << "Printing integers preceded by their base:" << endl
16       << showbase;
17
18    cout << x << endl; // print decimal value
19    cout << oct << x << endl; // print octal value
```

```
20    cout << hex << x << endl; // print hexadecimal value
21    return 0;
22 } // end main
```

```
Printing integers preceded by their base:
100
0144
0x64
```

شکل ۱۷-۱۵ دستکاری کننده استریم showbase.

۱۵-۷-۵ اعداد اعشاری، نماد علمی و ثابت (fixed, scientific)

دستکاری کننده‌های استریم **scientific** و **fixed** بر روی فرمت یا قالب‌بندی اعداد اعشاری کنترل دارند. دستکاری کننده **scientific** خروجی یک عدد اعشاری را مجبور می‌کند تا با فرمت علمی بنمایش درآید. دستکاری کننده **fixed** نیز یک عدد اعشاری را مجبور می‌کند تا به تعداد مشخص شده‌ای از ارقام در سمت راست نقطه اعشار به نماش درآورد. بدون استفاده از دستکاری کننده دیگری، مقدار یک عدد اعشاری تعیین کننده فرمت خروجی خواهد بود.

در برنامه شکل ۱۸-۱۵ یک عدد اعشاری در هر دو قالب ثابت و علمی با استفاده از **field** و **scientific** در خطوط 21 و 25 نشان داده شده است. امکان دارد فرمت توان در نماد علمی در میان کامپایلرهای مختلف با یکدیگر تفاوت داشته باشد.

```
1  // Fig. 15.18: Fig15_18.cpp
2  // Displaying floating-point values in system default,
3  // scientific and fixed formats.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7  using std::fixed;
8  using std::scientific;
9
10 int main()
11 {
12    double x = 0.001234567;
13    double y = 1.946e9;
14
15    // display x and y in default format
16    cout << "Displayed in default format:" << endl
17       << x << '\t' << y << endl;
18
19    // display x and y in scientific format
20    cout << "\nDisplayed in scientific format:" << endl
21       << scientific << x << '\t' << y << endl;
22
23    // display x and y in fixed format
24    cout << "\nDisplayed in fixed format:" << endl
25       << fixed << x << '\t' << y << endl;
26    return 0;
27 } // end main
```

```
Displayed in defualt format:
0.00123457      1.946e+009

Displayed in scientific format:
1.234567e-003   1.946000e+009

Displayed in fixed format:
0.001235        1946000000.000000
```

شکل ۱۸-۱۵ | مقادیر اعشاری در حالت نمایش عادی، علمی و ثابت شده.

۱۵-۷-۶ کنترل حروف بزرگ/کوچک (uppercase)

دستکاری کننده استریم **uppercase** یک حرف بزرگ **X** یا **E** را به همراه مقادیر هگزادسیمال یا با مقادیر اعشاری با نماد علمی به نمایش در می‌آورد (شکل ۱۹-۱۵). همچنین استفاده از دستکاری کننده **uppercase** سبب می‌شود تا تمام حروف موجود در یک مقدار هگزادسیمال تبدیل به حروف بزرگ شوند. بطور پیش‌فرض حروف موجود در مقادیر هگزادسیمال و توان در مقادیر اعشاری با نماد علمی بصورت کوچک ظاهر می‌شوند. برای از کار انداختن تنظیمات **uppercase** از **nouppercase** استفاده می‌شود.

```
1  // Fig. 15.19: Fig15_19.cpp
2  // Stream-manipulator uppercase.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6  using std::hex;
7  using std::showbase;
8  using std::uppercase;
9
10 int main()
11 {
12    cout << "Printing uppercase letters in scientific" << endl
13       << "notation exponents and hexadecimal values:" << endl;
14
15    // use std:uppercase to display uppercase letters; use std::hex and
16    // std::showbase to display hexadecimal value and its base
17    cout << uppercase << 4.345e10 << endl
18       << hex << showbase << 123456789  << endl;
19    return 0;
20 } // end main
```

```
Printing uppercase letters in scientific
notation exponents and hexadecimal valus:
4.345E+010
0x75BCD15
```

شکل ۱۹-۱۵ | دستکاری کننده استریم uppercase.

۱۵-۷-۷ قالب‌بندی بولی (boolalpha)

زمان C++ دارای نوع داده **bool** است که می‌توانند بصورت **false** یا **true** باشند و بعنوان جانشین مناسبی برای حالت قدیمی استفاده از صفر برای نشان دادن **false** و مقادیر غیرصفر برای نشان دادن **true** به شمار می‌آیند. یک متغیر بولی، صفر یا 1 را در حالت عادی چاپ می‌کند. با این وجود، می‌توانیم از دستکاری کننده استریم **boolalpha** برای تنظیم استریم خروجی در نمایش مقادیر بولی بصورت رشته‌ای "true" و "false" استفاده کنیم. دستکاری کننده استریم **noboolalpha** برای نمایش مقادیر بولی بفرم صحیح (0 یا مقدار غیر صفر، حالت پیش‌فرض) بکار گرفته می‌شود. برنامه شکل ۱۵-۲۰ به بررسی این دستکاری کننده استریم پرداخته است. خط 14 مقدار بولی را که در خط 11 با **true** تنظیم شده است را بصورت یک مقدار صحیح به نمایش در میآورد. خط 18 از دستکاری کننده **boolalpha** برای نمایش مقدار بولی بصورت رشته استفاده کرده است. سپس خطوط 22-21 مقدار بولی را تغییر داده و از **noboolalpha** استفاده کرده است، از اینرو خط 25 می‌تواند مقدار بولی را بعنوان یک مقدار صحیح به نمایش در آورد. در خط 29 از **boolalpha** برای نمایش مقدار بولی بصورت یک رشته استفاده شده است.

```
1  // Fig. 15.20: Fig15_20.cpp
2  // Demonstrating stream-manipulators boolalpha and noboolalpha.
3  #include <iostream>
4  using std::boolalpha;
5  using std::cout;
6  using std::endl;
7  using std::noboolalpha;
8
9  int main()
10 {
11    bool booleanValue = true;
12
13    // display default true booleanValue
14    cout << "booleanValue is " << booleanValue << endl;
15
16    // display booleanValue after using boolalpha
17    cout << "booleanValue (after using boolalpha) is "
18       << boolalpha << booleanValue << endl << endl;
19
20    cout << "switch booleanValue and use noboolalpha" << endl;
21    booleanValue = false; // change booleanValue
22    cout << noboolalpha << endl; // use noboolalpha
23
24    // display default false booleanValue after using noboolalpha
25    cout << "booleanValue is " << booleanValue << endl;
26
27    // display booleanValue after using boolalpha again
28    cout << "booleanValue (after using boolalpha) is "
29       << boolalpha << booleanValue << endl;
30    return 0;
31 } // end main
```

```
booleanValue is 1
booleanValue (after using boolalpha) is true

switch booleanValue and use noboolalpha

booleanValue is 0
booleanValue (after using boolalpha) is false
```

شکل ۲۰-۱۵ | دستکاری کننده‌های استریم boolalpha و noboolalpha.

۸-۷-۱۵ تنظیم و تنظیم مجدد وضعیت قالب‌بندی از طریق تابع عضو flags

در سرتاسر بخش ۷-۱۵ از انواع دستکاری کنندهاای استریم به منظور تغییر در صفات قالب‌بندی خروجی استفاده کردیم. اکنون بحث خود را متوجه نحوه باز گرداندن فرمت استریم خروجی به حالت یا وضعیت پیش‌فرض خود پس از اعمال دستکاری کننده‌ها می‌کنیم. تابع عضو **flags** بدون آرگومان، تنظیمات فرمت جاری را از نوع داده **fmtflags** برگشت می‌دهد (از کلاس **iso_base**)، که نشاندهنده وضعیت فرمت یا قالب‌بندی است. تابع عضو **flags** به همراه آرگومان **fmtflags** مبادرت به تنظیم فرمت وضعیت با توجه به آرگومان کرده و تنظیمات سابق را برمی‌گرداند. ممکن است مقدار تنظیم اولیه برگشتی توسط **flags** در سیستم‌های مختلف با هم تفاوت داشته باشند. در برنامه شکل ۲۱-۱۵ از تابع عضو **flags** برای ذخیره وضعیت فرمت اولیه استریم (خط 22)، سپس بازیابی فرمت اصلی استفاده شده است (خط 30).

```
1  // Fig. 15.21: Fig15_21.cpp
2  // Demonstrating the flags member function.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6  using std::ios_base;
7  using std::oct;
8  using std::scientific;
```

```
9  using std::showbase;
10
11 int main()
12 {
13    int integerValue = 1000;
14    double doubleValue = 0.0947628;
15
16    // display flags value, int and double values (original format)
17    cout << "The value of the flags variable is: " << cout.flags()
18       << "\nPrint int and double in original format:\n"
19       << integerValue << '\t' << doubleValue << endl << endl;
20
21    // use cout flags function to save original format
22    ios_base::fmtflags originalFormat = cout.flags();
23    cout << showbase << oct << scientific; // change format
24
25    // display flags value, int and double values (new format)
26    cout << "The value of the flags variable is: " << cout.flags()
27       << "\nPrint int and double in a new format:\n"
28       << integerValue << '\t' << doubleValue << endl << endl;
29
30    cout.flags( originalFormat ); // restore format
31
32    // display flags value, int and double values (original format)
33    cout << "The restored value of the flags variable is: "
34       << cout.flags()
35       << "\nPrint values in original format again:\n"
36       << integerValue << '\t' << doubleValue << endl;
37    return 0;
38 } // end main
```

```
The value of the flags variable is: 513
Print int and double in original format:
    1000    0.0947628

The value of the flags variable is: 012011
Print int and double in a new format:
    1750    9.476280e-002

The restored value of the flags variable is: 513
Print values in original format again:
1000    0.0947628
```

شکل ۲۱-۱۵ تابع عضو flags.

۸-۱۵ وضعیت خطا در استریم

می‌توان وضعیت یک استریم را با تست بیت‌های موجود در کلاس **ios_base** مشخص کرد. در برنامه شکل ۲۲-۱۵ نحوه تست این بیت‌ها نشان داده شده است.

بیت **eofbit** برای تنظیم یک استریم ورودی پس از مواجه شدن با انتهای فایل بکار گرفته می‌شود. برنامه می‌تواند با استفاده از تابع **eof** مشخص کند که آیا به انتهای فایل در استریم رسیده است یا خیر. فراخوانی

```
cin.eof()
```

اگر به انتهای فایل رسیده باشد مقدار **true** و در غیر اینصورت **false** برگشت می‌دهد.

بیت **failbit** برای یک استریم زمانیکه تنظیم می‌شود که یک خطای فرمت در استریم رخ داده باشد، مانند زمانیکه برنامه در حال وارد کردن مقادیر صحیح باشد و یک کاراکتر غیر عددی در استریم دیده شود. زمانیکه چنین خطای رخ دهد، کاراکترها مفقود نمی‌شوند. تابع عضو **fail** گزارشی در ارتباط با شکست یا عدم شکست عملیات تهیه می‌کند، بازیابی از چنین خطاهای امکان‌پذیر است.

بیت **badbit** زمانیکه خطای رخ دهد که باعث از دست رفتن داده شود، تنظیم می‌گردد (برای آن استریم). تابع عضو **bad** گزارشی از شکست یا عدم شکست عملیات تهیه می‌کند. معمولاً چنین شکست‌های غیرقابل بازیابی و جبران است.

```
1  // Fig. 15.22: Fig15_22.cpp
2  // Testing error states.
3  #include <iostream>
4  using std::cin;
5  using std::cout;
6  using std::endl;
7
8  int main()
9  {
10    int integerValue;
11
12    // display results of cin functions
13    cout << "Before a bad input operation:"
14       << "\ncin.rdstate(): " << cin.rdstate()
15       << "\n    cin.eof(): " << cin.eof()
16       << "\n   cin.fail(): " << cin.fail()
17       << "\n    cin.bad(): " << cin.bad()
18       << "\n   cin.good(): " << cin.good()
19       << "\n\nExpects an integer, but enter a character: ";
20
21    cin >> integerValue; // enter character value
22    cout << endl;
23
24    // display results of cin functions after bad input
25    cout << "After a bad input operation:"
26       << "\ncin.rdstate(): " << cin.rdstate()
27       << "\n    cin.eof(): " << cin.eof()
28       << "\n   cin.fail(): " << cin.fail()
29       << "\n    cin.bad(): " << cin.bad()
30       << "\n   cin.good(): " << cin.good() << endl << endl;
31
32    cin.clear(); // clear stream
33
34    // display results of cin functions after clearing cin
35    cout << "After cin.clear()" << "\ncin.fail(): " << cin.fail()
36       << "\ncin.good(): " << cin.good() << endl;
37    return 0;
38 } // end main
```

```
Before a bad input operation:
cin.rdstate(): 0
    cin.eof(): 0
   cin.fail(): 0
    cin.bad(): 0
   cin.good(): 1

Expects an integer, but enter a character: A
cin.rdstate(): 2
    cin.eof(): 0
   cin.fail(): 1
    cin.bad(): 0
   cin.good(): 0
After cin.clear()
cin.fail(): 0
cin.good(): 1
```

شکل ۲۲-۱۵ | تست وضعیت خطا.

اگر هیچ یک از بیت‌های **eofbit**، **failbit** یا **badbit** برای استریمی تنظیم نشده باشند، بیت **goodbit** تنظیم خواهد شد. تابع عضو **good** در صورتیکه توابع **bad**، **fail** و **eof** تماماً **false** باز گردانند، مبادرت به برگشت **true** می‌کند. عملیات I/O باید فقط با استریم‌های **good** کار کند.

تابع عضو **rdstate** مبادرت به بازگرداندن وضعیت خطا از استریم می‌کند. برای مثال با فراخوانی **cout.rdstate** وضعیت استریم برگشت داده خواهد شد، که سپس می‌تواند توسط یک عبارت **switch** به منظور بررسی **eofbit**، **badbit**، **failbit** و **goodbit** بکار گرفته شود.

تابع عضو **clear** برای اعاده کردن وضعیت استریم به حالت مناسب "good" بکار گرفته می‌شود. از اینرو است که I/O می‌تواند بر روی استریم ادامه یابد. آرگومان پیش‌فرض برای **clear** گزینه **goodbit** است، از اینرو عبارت

```
cin.clear();
```

مبادرت به ترخیص **cin** و تنظیم **goodbit** می‌کند. عبارت

```
cin.clear( iso::failbit );
```

مبادرت به تنظیم **failbit** می‌کند. امکان دارد برنامه‌نویس اینکار را به هنگام انجام عملیات ورودی بر روی **cin** با نوع داده تعریف شده از سوی کاربر و در مواجه شدن با یک مشکل انجام دهد. ممکن است نام **clear** چندان مناسب کاری که انجام می‌دهد نباشد، اما چنین است.

در برنامه شکل ۲۲-۱۵ به بررسی توابع عضو **rdstate**، **eof**، **fail**، **bad**، **good** و **clear** پرداخته شده است. تابع عضو **operator!** از کلاس **basic_ios** در صورتیکه **badbit**، **failbit** یا هر دو تنظیم شده باشند، مقدار **true** برگشت می‌دهد. تابع عضو **operator void *** در صورتیکه **badbit**، **failbit** یا هر دو تنظیم شده باشد مقدار **false** برگشت می‌دهد. این توابع به هنگام پردازش فایل سودمند هستند.

۱۵-۹ پیوند استریم خروجی با استریم ورودی

معمولاً برنامه‌های تعاملی درگیر یک **istream** برای ورودی و یک **ostream** برای خروجی هستند. زمانیکه یک پیغام بر روی صفحه ظاهر می‌شود، کاربر با وارد کردن داده مناسب به آن پاسخ می‌دهد. بدیهی است که این پیغام باید قبل از عملیات ورودی ظاهر شود. با بافر کردن خروجی، خروجی فقط زمانیکه بافر پر شود، یا بصورت صریح توسط برنامه یا بطور اتوماتیک در انتهای برنامه خالی شود، ظاهر خواهد شد. زبان C++ تابع عضو **tie** را برای همزمان کردن (یعنی با یکدیگر پیوند داشتن) عملیات یک **istream** و یک **ostream** در اختیار گذاشته تا مطمئن شویم که خروجی‌ها قبل از ورودی‌های متعاقب آنها ظاهر خواهند شد. در فراخوانی

```
cin.tie( &cout );
```

cout (یک **ostream**) به **cin** (یک **istream**) پیوند زده می‌شود. در واقع این نوع از فراخوانی اضافی است، چرا که C++ این نوع عملیات را بصورت اتوماتیک برای ایجاد یک محیط استاندارد I/O برای

کاربر فراهم می‌آورد. با این همه، کاربر می‌تواند سایر نوع‌های **isteam/ostream** را بصورت صریح به یکدیگر پیوند دهد. برای گشودن یک استریم ورودی، **inputStream**، از یک استریم خروجی، از فراخوانی زیر استفاده می‌شود:

```
inputStream.tie( 0 );
```

فصل شانزدهم

رسیدگی به استثناء

اهداف

- استثناء چیست و در چه مواقعی از آنها استفاده می‌کنیم.
- استفاده از try، catch و throw برای تشخیص، رسیدگی و نمایان ساختن استثناها.
- پردازش استثناهای غیرمنتظره و گرفتار نشده.
- اعلان کلاس‌های استثناء.
- چگونه باز کردن پشته می‌تواند استثناهای گرفتار نشده در یک قلمرو را در قلمرو دیگری گرفتار سازد.
- رسیدگی به واماندگی new.
- استفاده از auto_ptr برای اجتناب از فقدان حافظه.
- آشنایی با سلسله مراتب استاندارد استثناء.

رئوس مطالب

۱۶-۱ مقدمه

در این فصل، در مورد رسیدگی به استثناها بحث می‌کنیم. یک استثناء دلالت بر وجود مشکلی دارد که در زمان اجرای برنامه رخ می‌دهد. نام "استثناء" از این حقیقت بدست آمده که، اگر چه مشکل می‌تواند همیشه رخ دهد، اما استثناء بندرت رخ می‌دهد. اگر "قاعده‌ای" بر این اصل استوار است که عبارتی بطرز صحیح اجرا شود، اما رویدادی موجب رخ دادن مشکلی گردد پس "استثنای در این قاعده" وجود دارد.

رسیدگی به استثناء به برنامه‌نویسان امکان می‌دهد تا برنامه‌هایی ایجاد کنند که قادر به برطرف کردن (یا رسیدگی) استثناءها هستند. در بسیاری از موارد، رسیدگی به یک استثناء به برنامه امکان می‌دهد در صورت عدم برخورد با مشکلی به اجرای خود ادامه دهد. با این همه، مشکلات جدی واساسی می‌توانند مانع از اجرای عادی برنامه شوند در چنین حالاتی برنامه باید مشکل را به کاربر اطلاع داده و با یک روش کنترل شده به مشکل خاتمه دهد. ویژگی‌های مطرح شده در این فصل برنامه‌نویسان را قادر به نوشتن برنامه‌های واضح، پایدار و مقاوم در برابر خطا می‌کنند.

قالب و جزئیات روش رسیدگی به خطا در C++ مبتنی بر تحقیقات Andrew koenig و Bjarne Stroustrup در مقاله‌ای بنام "Exception Handling for C++ (revised)" است.

این فصل با معرفی مفاهیم رسیدگی به استثناء و توصیف تکنیک‌های پایه در این زمینه آغاز می‌شود.

۲-۱۶ مفهوم رسیدگی به استثناء

منطق یک برنامه، تست مکرر شرایط برای تعیین نحوه عملکرد اجرای برنامه است. به شبه کد زیر توجه کنید:

Perform a task *(انجام یک وظیفه)*

If the preceding task did not execute correctly *(اگر وظیفه قبلی به درستی اجرا نشده باشد)*

Perform error processing *(فرآیند پردازش خطا انجام گیرد)*

Perform next task *(انجام وظیفه بعدی)*

If the preceding task did not execute carrectly *(اگر وظیفه قبلی بدرستی اجرا نشده باشد)*

Perform error processing *(فرآیند پردازش خطا انجام گیرد)*

...

در این شبه کد، کار با انجام یک وظیفه آغاز می‌شود. سپس تست می‌شود که وظیفه بدرستی به انجام رسیده است یا خیر. اگر چنین نباشد، فرآیند خطا به اجرا در می‌آید. در غیر اینصورت، برنامه با اجرای وظیفه بعدی بکار خود ادامه می‌دهد. اگر چه این فرم از رسیدگی به خطا کار می‌کند، اما کاربرد چنین منطقی از رسیدگی به خطا می‌تواند سبب پیچیده شدن ساختار برنامه شده و قرائت، اصلاح، نگهداری و دیباگ آنرا با مشکل مواجه کند. این امر در مورد برنامه‌های کاربردی بزرگ مصادق بیشتری دارد. در واقع، اگر تعدادی از مشکلات نهانی بندرت رخ دهند، چنین منطقی از برنامه و رسیدگی به خطا می‌تواند سبب کاهش کارایی برنامه شود، چرا که برنامه مجبور است تا به بررسی شرایط بیشتری بپردازد تا تعیین کند که آیا وظیفه مورد نظر می‌تواند اجرا شود یا خیر.

ویژگی رسیدگی به استثناء به برنامه‌نویس امکان می‌دهد تا اقدام به حذف کد رسیدگی کننده به خطا از "خط اصلی" در مسیر اجرای برنامه نماید. با این عمل وضوح و کارایی برنامه افزایش می‌یابد. برنامه‌نویسان می‌تواند در مورد اینکه می‌خواهند به کدام استثناء رسیدگی کنند، تصمیم بگیرند، (تمام انواع استثناءها، تمام استثناءها از یک نوع مشخص یا تمام استثناءها از یک گروه مرتبط با یکدیگر). چنین انعطافی احتمال نادیده گرفته شدن خطاها را کاهش داده و از اینرو پایداری برنامه افزایش می‌یابد.

تست و خطایابی

رسیدگی به خطا تحمل‌پذیری برنامه‌ها را در مقابل خطا افزایش می‌دهد.

برنامه‌نویسی ایده‌ال

از بکارگیری رسیدگی به استثناء در مواردی بجز رسیدگی به خطا اجتناب کنید، چرا که چنین استفاده‌ای

می‌تواند وضوح برنامه را کاهش دهد.

زمانیکه از زبان‌های برنامه‌نویسی استفاده می‌شود که از ویژگی رسیدگی به استثناء پشتیبانی نمی‌کنند، غالباً برنامه‌نویسان نوشتن کدهای پردازش خطا را به تعویق می‌اندازند و گاهی اوقات افزودن آنها را به برنامه فراموش می‌کنند. در نتیجه توانایی و پایداری نرم‌افزار تولیدی کاهش می‌یابد. C++ برنامه‌نویسان را قادر می‌سازد تا به روش مناسبی رسیدگی به استثناء را در پروژه‌های خود وارد سازند.

مهندسی نرم‌افزار

در گذشته، برنامه‌نویسان از تکنیک‌های متفاوتی برای پیاده‌سازی کدهای پردازش خطا استفاده می‌کردند. در حالیکه رسیدگی به استثناء ارائه کننده یک تکنیک واحد و متحد برای پردازش خطا عرضه می‌کند.

کارایی

زمانیکه هیچ استثنای رخ ندهد، کد رسیدگی به استثناء، در کارایی برنامه تأثیر منفی نخواهد داشت.

کارایی

از ویژگی استثناء فقط در مورد مشکلاتی استفاده کنید که بندرت رخ می‌دهند.

۳-۱۶ مثال: رسیدگی به خطای تقسیم بر صفر

اجازه دهید تا به بررسی یک مثال ساده از رسیدگی به خطا بپردازیم (شکل‌های ۱-۱۶ و ۲-۱۶). هدف از این مثال اجتناب از یک مشکل رایج در محاسبات یعنی عملیات تقسیم بر صفر است. در C++، تقسیم بر صفر با استفاده از محاسبات صحیح سبب می‌شود تا برنامه بصورت نابهنگام خاتمه یابد. در محاسبات اعشاری، تقسیم بر صفر امکان‌پذیر است، که حاصل آن بی‌نهایت منفی یا مثبت است که بصورت INF یا -INF به نمایش در می‌آید.

در این مثال، تابعی بنام **quotient** (خارج قسمت) تعریف می‌کنیم که دو ورودی از نوع صحیح از سوی کاربر دریافت و اولین پارامتر **int** خود را به دومین پارامتر **int** تقسیم می‌کند. قبل از مبادرت به تقسیم، تابع مقدار پارامتر اول را به نوع **double** تبدیل می‌کند. سپس مقدار دومین پارامتر به نوع **double** ارتقا داده می‌شود تا محاسبه صورت گیرد. بنابر این تابع **quotient** در واقع تقسیم را با استفاده از دو مقدار **double** انجام می‌دهد و حاصل آن نیز یک مقدار **double** خواهد بود.

اگرچه تقسیم بر صفر در محاسبات اعشاری امکان‌پذیر است، اما برای برآورد کردن هدف این مثال، فرض می‌کنیم هر گونه تقسیم بر صفر یک خطا محسوب می‌شود. بنابر این، تابع **quotient** پارامتر دوم خود را تست می‌کند تا مطمئن گردد که صفر نباشد، قبل از اینکه اجازه تقسیم داده شود. اگر پارامتر دوم، صفر

باشد، تابع با استفاده از یک استثنا به فراخوان نشان می‌دهد که یک مشکل رخ داده است. سپس فراخوان (در این مثال **main**) می‌تواند این استثنا را پردازش کرده و به کاربر اجازه دهد دو مقدار جدید را قبل از فراخوانی مجدد **quotient** وارد سازد. به این روش، برنامه می‌تواند به اجرای خود ادامه دهد حتی پس از اینکه یک مقدار اشتباه وارد شده باشد، بنابر این برنامه از استحکام بیشتری برخوردار خواهد شد.

این مثال، متشکل از دو فایل است: *DivideByZeroException.h* یک کلاس استثنا تعریف می‌کند که نشاندهنده نوع مشکلی است که امکان دارد در این مثال رخ دهد (شکل ۱-۱۶) و *fig16_02.cpp* که تعریف کننده تابع **quotient** و تابع **main** است که آنرا فراخوانی می‌کند. تابع main حاوی کدی است که به توصیف رسیدگی به استثنا می‌پردازد.

تعریف یک کلاس استثنا برای نمایش نوع مشکلی که اتفاق افتاده است

برنامه شکل ۱-۱۶ تعریف کننده کلاس **DivideByZeroException** بعنوان یک کلاس مشتق شده از کلاس **runtime_error** کتابخانه استاندارد است (تعریف شده در فایل سرآیند <**stdexcept**>) کلاس **runtime_error** که از کلاس **exception** مشتق شده است (تعریف شده در فایل سرآیند <**exception**>)، کلاس مبنا استاندارد C++ در عرضه خطاهای زمان اجرا است. کلاس **exeception** کلاس مبنا استاندارد C++ برای تمام استثناها می‌باشد (در بخش ۱۳-۱۶ با این کلاس بیشتر آشنا خواهید شد). یک نمونه از کلاس استثنا که از کلاس **runtime_error** مشتق می‌شود، فقط یک سازنده تعریف می‌کند (همانند، خطوط 13-12) که یک رشته پیغام خطا را به سازنده کلاس مبنا **runtime_error** ارسال می‌نماید. هر کلاس استثنا که مستقیماً یا غیرمستقیم از **exception** مشتق می‌شود حاوی یک تابع مجازی یا **virtual** بنام **what** است که، یک پیغام خطا از شی استثنا برگشت می‌دهد. دقت کنید که مجبور نیستید از یک کلاس استثنا رایج عمل مشتق را انجام دهید، همانند **DividoeByZeroException** از کلاس‌های استثناء استاندارد تدارک دیده شده توسط C++. با این همه، انجام اینکار به برنامه‌نویسان امکان استفاده از تابع مجازی **what** را برای بدست آوردن پیغام مناسبی از خطای رخ داده، فراهم می‌آورد. در برنامه شکل ۲-۱۶ از یک شی کلاس **DivideByZeroExeception** استفاده کرده‌ایم تا نشان دهنده زمان اقدام به عملیات تقسیم بر صفر باشد.

```
1  // Fig. 16.1: DivideByZeroException.
2  // Class DivideByZeroException definition.
3
4  #include <stdexcept> // stdexcept header file contains runtime_error
5  using std::runtime_error; // standard C++ library class runtime_error
6
7  // DivideByZeroException objects should be thrown by functions
8  // upon detecting division-by-zero exceptions
9  class DivideByZeroException : public runtime_error
10  {
11 public:
12    // constructor specifies default error message
13    DivideByZeroException::DivideByZeroException()
```

```
14       : runtime_error( "attempted to divide by zero" ) {}
15 }; // end class DivideByZeroException
```

شکل ۱-۱۶ | تعریف کلاس DivideByZeroException.

```
1  // Fig. 16.2: Fig16_02.cpp
2  // A simple exception-handling example that checks for
3  // divide-by-zero exceptions.
4  #include <iostream>
5  using std::cin;
6  using std::cout;
7  using std::endl;
8  #include "DivideByZeroException.h" // DivideByZeroException class
9
10 // perform division and throw DivideByZeroException object if
11 // divide-by-zero exception occurs
12 double quotient( int numerator, int denominator )
13 {
14    // throw DivideByZeroException if trying to divide by zero
15    if ( denominator == 0 )
16       throw DivideByZeroException(); // terminate function
17
18    // return division result
19    return static_cast< double >( numerator ) / denominator;
20 } // end function quotient
21
22 int main()
23 {
24    int number1; // user-specified numerator
25    int number2; // user-specified denominator
26    double result; // result of division
27
28    cout << "Enter two integers (end-of-file to end): ";
29
30    // enable user to enter two integers to divide
31    while ( cin >> number1 >> number2 )
32    {
33       // try block contains code that might throw exception
34       // and code that should not execute if an exception occurs
35       try
36       {
37          result = quotient( number1, number2 );
38          cout << "The quotient is: " << result << endl;
39       } // end try
40
41       // exception handler handles a divide-by-zero exception
42       catch ( DivideByZeroException &divideByZeroException )
43       {
44          cout << "Exception occurred: "
45             << divideByZeroException.what() << endl;
46       } // end catch
47
48       cout << "\nEnter two integers (end-of-file to end): ";
49    } // end while
50
51    cout << endl;
52    return 0; // terminate normally
53 } // end main
```

```
Enter two integers (end-of-file to end): 100 7
The quotient is: 14.28.57

Enter two integers (end-of-file to end): 100 0
Exception occurred: attempted to divide by zero

Enter two integers (end-of-file to end):^z
```

شکل ۲-۱۶ |مثالی از رسیدگی به استثنا که به هنگام تقسیم بر صفر استثناهای را راه‌اندازی می‌کند.

شرح رسیدگی به استثنا

در برنامه شکل ۲-۱۶ از رسیدگی به استثنا برای پوشاندن کدی که می‌تواند سبب‌ساز استثنا «تقسیم بر صفر» شود و برای رسیدگی به آن استفاده شده است. برنامه به کاربر امکان می‌دهد تا دو مقدار صحیح وارد سازد و آنها را بعنوان آرگومان‌های به تابع **quotient** ارسال می‌کند (خطوط 21-13). این تابع عدد اول (**numerator**) را به عدد دوم (**denominator**) تقسیم می‌کند. با فرض اینکه کاربر مقدار صفر را برای مقسوم علیه وارد نکرده باشد، تابع **quotient** نتیجه تقسیم را برگشت می‌دهد. با این وجود، اگر کاربر مبادرت به وارد کردن صفر بعنوان مقسوم علیه کند، تابع **quotient** یک استثنا به راه می‌اندازد. در خروجی نمونه این برنامه، دو خط ابتدائی نمایشی از موفقیت‌آمیز بودن محاسبه دارند، و دو خط بعدی نمایشی از واماندگی محاسبه در زمان به تقسیم بر صفر هستند. زمانیکه استثنا رخ می‌دهد، برنامه به کاربر اشتباه رخ داده را اطلاع می‌دهد و از وی می‌خواهد دو مقدار صحیح جدید وارد سازد. پس از بررسی کد، به بررسی ورودی‌های کاربر و جریان کنترل برنامه می‌پردازیم که حاصل آن این خروجی‌ها است.

احاطه کردن کد در بلوک try

برنامه با اعلان وارد کردن دو عدد صحیح به برنامه شروع می‌شود. مقادیر صحیح در شرط حلقه **while** وارد می‌شوند (خط 32). پس از اینکه کاربر مقادیری را بعنوان صورت کسر و مقسوم علیه (مخرج کسر) وارد کرد، کنترل برنامه، رهسپار بدنه حلقه می‌شود (خطوط 50-33).

خط 38 این مقادیر را به تابع **quotient** ارسال می‌کند (خطوط 21-13)، که عملیات تقسیم را انجام داده و نتیجه‌ای برگشت می‌دهد یا یک استثنا به راه می‌افتد (یعنی یک خطا رخ داده است) که نشاندهنده تقسیم بر صفر است. رسیدگی به خطا ابزاری برای تابع است که خطای را تشخیص می‌دهد که قادر به رسیدگی به آن نیست.

زبان C++ با تدارک دیدن بلوک‌های **try** قادر به رسیدگی به استثناء است. یک بلوک **try** متشکل از کلمه کلیدی **try** و بدنبال آن براکت‌ها (**{ }**) است که یک بلوک از کد را تعریف می‌کنند که احتمال دارد استثناهای در آنجا رخ دهد. بلوک **try** عباراتی را که احتمال دارد استثناهای را سبب شوند و عباراتی که بایستی در هنگام رخ دادن استثنا از آنها عبور شود، احاطه می‌کند.

به بلوک **try** در خطوط 40-36 توجه کنید که فراخوانی تابع **quotient** و عبارتی که نتیجه خطا را به نمایش در می‌آورد، احاطه کرده است. در این مثال، بدلیل اینکه احضار تابع **quotient** در خط 38 می‌تواند یک استثنا به راه بیاندازد، فراخوانی این تابع را در بلوک **try** قرار داده‌ایم. احاطه کردن عبارت خروجی (خط 39) در بلوک **try** ما را مطمئن می‌سازد که خروجی فقط در صورتی که تابع **quotient** نتیجه‌ای برگشت دهد، به نمایش درخواهد آمد.

تعریف رسیدگی catch برای پردازش DivideByZeroException

استثناها توسط رسیدگی کننده‌های **catch** پردازش می‌شوند، که استثناها را گرفته و پردازش می‌کنند. حداقل بایستی یک رسیدگی کننده **catch** بلافاصله بدنبال هر بلوک **try** آورده شود (خطوط 47-43). هر رسیدگی کننده **catch** با کلمه کلیدی **catch** و پارامتر استثنا در درون پرانتزها که نشاندهنده نوع استثنای است که **catch** می‌تواند پردازش نماید، آغاز می‌شود (در این مثال ***DivideByZeroException***).

زمانیکه یک استثنا در بلوک **try** رخ می‌دهد، رسیدگی کننده **catch** براساس نوع استثنا رخ داده شروع بکار می‌کند. اگر پارامتر استثنا شامل نام یک پارامتر اختیاری باشد، رسیدگی کننده **catch** می‌تواند از آن نام پارامتر برای تعامل با شی استثنا گرفتار شده در بدنه **catch** استفاده کند، که با براکت‌ها (**{}**) حدود آن تعیین شده است. معمولاً یک رسیدگی کننده **catch** یک گزارش خطا به کاربر عرضه کرده، آنرا در یک فایل واقعه (*log*) ثبت و برنامه را بطرز خوبی پایان داده یا سعی می‌کند با اعمال یک استراتژی جایگزین وظیفه شکست خورده را به انجام برساند. در این مثال، رسیدگی کننده **catch** فقط یک گزارش ساده عرضه می‌کند و نشان می‌دهد که کاربر اقدام به تقسیم بر صفر کرده است. سپس برنامه به کاربر اعلان می‌کند تا دو مقدار جدید وارد سازد.

خطای برنامه‌نویسی

قرار دادن کد مابین یک بلوک ***try*** *و رسیدگی کننده‌های* ***catch*** *متناظر، خطای نحوی است.*

خطای برنامه‌نویسی

هر رسیدگی کننده ***catch*** *فقط می‌تواند یک پارامتر داشته باشد، استفاده از کاما برای افزودن پارامتر، خطای نحوی است.*

مدل خاتمه‌دهی رسیدگی به استثنا

اگر یک استثنا بعنوان نتیجه یک عبارت در یک بلوک **try** رخ دهد، بلوک **try** به پایان می‌رسد (یعنی بلافاصله خاتمه می‌یابد). سپس برنامه جستجوی برای اولین رسیدگی کننده **catch** انجام می‌دهد که بتواند نوع استثنا رخ داده را پردازش نماید. برنامه، **catch** مناسب را با مقایسه نوع استثنا رخ داده با نوع هر پارامتر استثنا در هر **catch** انتخاب می‌کند. زمانیکه مطابقتی یافت شد، کد موجود در رسیدگی کننده **catch** مورد نظر اجرا می‌گردد. زمانیکه پردازش رسیدگی کننده **catch** با رسیدن به براکت بسته سمت راست (**}**) به پایان رسید، به آن استثنا رسیدگی شده است و متغیرهای محلی تعریف شده در رسیدگی کننده **catch** (شامل پارامتر **catch**) از قلمرو خارج می‌شوند. کنترل برنامه به نقطه‌ای که استثنا در آنجا رخ داده بود برگشت داده نمی‌شود (بنام نقطه راه‌اندازی یا پرتاب شناخته می‌شود)، چرا که بلوک **try** به پایان رسیده است. کنترل به اولین عبارت (خط 49) پس از آخرین رسیدگی کننده **catch** که بدنبال **try** آمده است، منتقل می‌شود. این مدل بعنوان *مدل خاتمه رسیدگی به استثنا* شناخته می شود. همانند هر بلوکی از کد، زمانیکه بلوک **try** خاتمه می‌یابد، متغیرهای محلی تعریف شده در آن بلوک از قلمرو خارج می‌شوند.

اگر بلوک **try** با موفقیت اجرای خود را کامل کند (یعنی بدون رخ دادن استثنا در بلوک **try**)، برنامه رسیدگی کننده‌های **catch** را نادیده گرفته، و کنترل برنامه با اولین عبارت پس از آخرین **catch** پس از آن بلوک **try** ادامه می‌یابد. اگر هیچ استثنای در یک بلوک **try** رخ ندهد، برنامه رسیدگی کننده‌های **catch** برای آن بلوک را نادیده خواهد گرفت.

اگر برای استثنائی که در یک بلوک **try** رخ داده، هیچ رسیدگی کننده **catch** مطابق با آن پیدا نشود، یا اگر استثنا در عبارتی رخ دهد که در یک بلوک **try** وجود ندارد، تابعی که حاوی آن عبارت است، بلافاصله خاتمه یافته و برنامه مبادرت به یافتن یک بلوک **try** احاطه کننده در تابع فراخوانی شده می‌کند. این فرآیند با نام *stack unwinding* یا *باز کردن پشته* شناخته می‌شود و در بخش ۸-۱۶ به بررسی آن خواهیم پرداخت.

جریان کنترل برنامه زمانیکه کاربر برای مخرج یک مقدار غیر صفر وارد می‌کند

با فرض اینکه کاربر مقدار 100 را برای صورت و 7 را برای مخرج وارد کرده باشد، به جریان کنترل توجه کنید (یعنی، دو خط ابتدایی در خروجی برنامه شکل ۲-۱۶). در خط 16، تابع **quotient** تعیین می‌کند که **denominator** برابر صفر نیست، از اینرو خط 20 عملیات تقسیم را انجام داده و نتیجه (14.2857) را به خط 38 بعنوان یک نوع **double** برگشت می‌دهد. سپس کنترل برنامه بصورت ترتیبی از خط 38 ادامه می‌یابد، بنابر این خط 39 نتیجه تقسیم را به نمایش در آورده و خط 40 انتهای بلوک **try** است. بدلیل اینکه بلوک **try** بطور موفقیت‌آمیز کامل شده و استثنای به وجود نیامده است، برنامه عبارات موجود در درون رسیدگی کننده **catch** را به اجرا در نیاورده (خطوط 47-43)، و کنترل با خط 49 ادامه می‌یابد که به کاربر اعلان می‌کند دو عدد صحیح وارد سازد.

جریان کنترل برنامه زمانیکه کاربر برای مخرج صفر وارد می‌کند

اکنون اجازه دهید به حالت جالبی بپردازیم که در آن کاربر برای صورت عدد 100 و برای مخرج عدد صفر وارد نماید (یعنی خطوط سوم و چهارم از خروجی شکل ۲-۱۶). در خط 16، **quotient** تعیین می‌کند که مخرج برابر صفر است، که دلالت بر اقدام تقسیم بر صفر دارد. خط 17 یک استثنا به راه می‌اندازد که آنرا بصورت یک شی از کلاس **DivideByZeroException** نشان داده‌ایم (شکل ۱-۱۶).

به استثنا به راه افتاده دقت کنید، خط 17 از کلمه کلیدی **throw** و بدنبال آن یک عملوند که نشاندهنده نوع استثناء به راه افتاده است، استفاده کرده است. معمولاً عبارت **throw** از یک عملوند استفاده می‌کند (در بخش ۵-۱۶، در ارتباط با استفاده از عبارات **throw** که عملوندی ندارند صحبت خواهیم کرد). عملوند یک **throw** می‌تواند از هر نوعی باشد. اگر عملوند یک شی باشد، آنرا یک شی استثنا می‌نامیم که در این مثال، شی استثنا یک شی از نوع **DivideByZeroException** است. با این همه، یک عملوند

throw می‌تواند با مقادیر دیگری در نظر گرفته شود، همانند مقدار در یک عبارت (مثال **throw x>5**) یا مقداری از یک **int** (مثال، **throw 5**). مثال‌های مطرح شده در این فصل انحصاراً بر روی شی‌های استثنا متمرکز هستند.

بعنوان بخشی از راه‌اندازی یک استثناء، عملوند **throw** ایجاد شده و برای مقداردهی اولیه پارامتر در رسیدگی کننده **catch** بکار گرفته می‌شود، که در مورد آن صحبت کوتاهی خواهیم کرد. در این مثال، عبارت **throw** در خط 17 یک شی از کلاس **DivideByZeroException** ایجاد می‌کند. زمانیکه خط 17 استثنا را به راه می‌اندازد، تابع **quotient** بلافاصله از صحنه خارج می‌شود. بنابر این، خط 17 مبادرت به راه‌اندازی استثنا قبل از اینکه **quotient** بتواند تقسیم موجود در خط 20 را به انجام برساند، می‌کند. این رفتار یکی از صفات اصلی رسیدگی به استثنا است: تابع بایستی یک استثنا را قبل از اینکه خطایی فرصت رخ دادن پیدا کند، راه‌اندازی نماید.

بدلیل اینکه تصمیم به احاطه کردن احضار تابع **quotient** در بلوک **try** گرفته‌ایم (خط 38)، کنترل برنامه وارد رسیدگی کننده **catch** می‌شود (خطوط 47-43) که بلافاصله بدنبال بلوک **try** آمده است. این رسیدگی کننده **catch** همانند یک رسیدگی کننده استثنا برای استثنا تقسیم بر صفر خدمت می‌کند. بطور کلی، زمانی که یک استثنا در درون یک بلوک **try** به راه می‌افتد، استثنا توسط یک رسیدگی کننده **catch** گرفتار می‌شود که نوع آن با نوع استثنا مطابقت دارد. در این برنامه، رسیدگی کننده **catch** تصریح می‌کند که شی **DivideByZeroException** را گرفتار ساخته است، این نوع مطابقت با نوع شی به راه افتاده در تابع **quotient** دارد. در واقع رسیدگی کننده **catch** مراجعه به شی **DivideByZeroException** ایجاد شده توسط عبارت **throw** تابع **quotient** را گرفتار می‌سازد.

بدنه رسیدگی **catch** در خطوط 46-45 پیغام خطای متناسب برگشتی از فراخوانی تابع **what** از کلاس مبنای **runtime_error** را به نمایش در می‌آورد. این تابع رشته‌ای برگشت می‌دهد که سازنده **DivideByZeroException** (خطوط 18-12 ازشکل ۱-۱۶) به سازنده کلاس مبنای **runtime_error** ارسال می‌کند.

۱۶-۴ زمان استفاده از رسیدگی به استثنا

رسیدگی به استثنا برای پردازش خطاهای همگام طراحی شده است، که در زمان اجرای یک عبارت رخ می‌دهند. مثال‌های رایج از این قبیل خطاها عبارتند از خارج شدن شاخص آرایه از مرزها، سرریز محاسباتی، تقسیم بر صفر، پارامترهای اشتباه و عدم اخذ موفقیت‌آمیز حافظه (به علت فقدان حافظه). رسیدگی به استثنا برای پردازش خطاهای مرتبط با رویدادهای ناهمگام طراحی نشده است (همانند عملیات

I/O دیسک، ارسال پیغام در شبکه، کلیک‌های ماوس و ضربه کلید)، که بصورت موازی و مستقل از جریان کنترل برنامه رخ می‌دهند.

همچنین مکانیزم رسیدگی به استثنا برای پردازش مسائلی که در زمان تعامل برنامه با عناصر نرم‌افزاری همانند توابع عضو، سازنده‌ها، نابود کننده‌ها و کلاس رخ می‌دهند، مناسب است.

معمولاً برنامه‌های کاربردی پیچیده، متشکل از کامپونت‌های نرم‌افزاری از پیش تعریف شده و کامپونت‌های خاص برنامه هستند که از کامپونت‌های از پیش تعریف شده استفاده می‌کنند. زمانیکه یک کامپونت از پیش تعریف شده با مشکلی مواجه می‌شود، آن کامپونت نیاز به مکانیزمی برای بیان مشکل با کامپونت خاص برنامه دارد، در واقع کامپونت از پیش تعریف شده اطلاعی از اینکه برنامه چگونه مشکل رخ داده را پردازش می‌کند، ندارد.

۱۶-۵ راه‌اندازی مجدد استثنا

امکان دارد که یک رسیدگی کننده به استثنا، به محض دریافت یک استثنا، تصمیم بگیرد که نمی‌تواند استثنا را پردازش کند یا می‌تواند اندکی از آنرا پردازش نماید. در چنین مواردی رسیدگی کننده می‌تواند رسیدگی به استثنا را به رسیدگی کننده استثنا دیگری تسلیم نماید (یا بخشی از آنرا). در هر دو مورد، رسیدگی کننده اینکار را با راه‌اندازی مجدد استثنا از طریق عبارت زیر به انجام می‌رساند

```
throw;
```

صرفنظر از اینکه یک رسیدگی کننده بتواند استثنایی را پردازش کند یا نه، رسیدگی کننده می‌تواند برای پردازش استثنا در خارج از رسیدگی کننده، آنرا مجدداً راه‌اندازی کند. بلوک **try** بعدی راه‌اندازی مجدد استثنا را تشخیص می‌دهد، که یک **catch** قرار گرفته پس از بلوک **try** مبادرت به رسیدگی به استثنا می‌کند.

برنامه شکل ۳-۱۶ به بررسی راه‌اندازی مجدد یک استثنا پرداخته است. در بلوک **try** خطوط 37-32، خط 35 تابع **throwException** را فراخوانی می‌کند (خطوط 27-11). تابع **throwException** نیز حاوی یک بلوک **try** است (خطوط 18-14) که عبارت throw در خط 17 یک نمونه از کلاس کتابخانه استاندارد **exception** را راه‌اندازی می‌کند. رسیدگی کننده **catch** این تابع در خطوط 24-19 این استثنا را گرفته، پیغام خطا چاپ کرده (خطوط 22-21) و استثنا را مجدداً راه‌اندازی می‌کند (خط 23). با اینکار تابع **throwException** خاتمه یافته و کنترل به خط 35 در بلوک **try…catch** در **main** باز می‌گردد. بلوک **try** خاتمه یافته (از اینرو خط 36 اجرا نمی‌شود) و رسیدگی کننده **catch** در **main** این استثنا را گرفته (خطوط 41-38) و پیغام خطا را چاپ می‌کند (خط 40).

```
1  // Fig. 16.3: Fig16_03.cpp
2  // Demonstrating exception rethrowing.
3  #include <iostream>
```

```
4  using std::cout;
5  using std::endl;
6
7  #include <exception>
8  using std::exception;
9
10 // throw, catch and rethrow exception
11 void throwException()
12 {
13    // throw exception and catch it immediately
14    try
15    {
16       cout << "  Function throwException throws an exception\n";
17       throw exception(); // generate exception
18    } // end try
19    catch ( exception & ) // handle exception
20    {
21       cout << "  Exception handled in function throwException"
22          << "\n  Function throwException rethrows exception";
23       throw; // rethrow exception for further processing
24    } // end catch
25
26    cout << "This also should not print\n";
27 } // end function throwException
28
29 int main()
30 {
31    // throw exception
32    try
33    {
34       cout << "\nmain invokes function throwException\n";
35       throwException();
36       cout << "This should not print\n";
37    } // end try
38    catch ( exception & ) // handle exception
39    {
40       cout << "\n\nException handled in main\n";
41    } // end catch
42
43    cout << "Program control continues after catch in main\n";
44    return 0;
45 } // end main
```

```
main invokes function throwException
 Function throwException throws an exception
 Exception handled in function throwException
 Function throwException rethrows exception

 Exeption hendled in main
 Program control continues after catch in main
```

شکل ۳-۱٦ | راه‌اندازی مجدد استثنا.

۱٦-٦ مشخصات استثنا

یکی از مشخصات اختیاری استثنا لیستی از استثناها است که یک تابع می‌تواند آنها را به جریان در آورد. برای مثال، به اعلان تابع زیر توجه کنید

```
int someFunction( double value )
      throw( ExceptionA, ExceptionB, ExceptionC)
{
   //function body
}
```

در این تعریف، مشخصات استثنا، که با کلمه کلیدی **throw** شروع شده و بدنبال آن پرانتزهای با لیست پارامتری تابع قرار دارند، بر این نکته دلالت دارد که تابع **someFunction** می‌تواند استثناهای از نوع

ExceptionA، **ExecptionB** و **ExceptionC** راه‌اندازی کند. تابع فقط می‌تواند استثناهای از نوع‌های مشخص شده در مشخصات را راه‌اندازی کند یا استثناهای از هر نوع مشتق شده از این نوع‌ها. اگر تابع استثنائی را سبب شود که متعلق به یک نوع مشخص شده باشد، تابع **unexpected** فراخوانی می‌شود، که معمولاً به برنامه خاتمه می‌دهد.

تابعی که مشخصات استثنا برای آن تدارک دیده نشده است، می‌تواند هر نوع استثنایی را به راه بیاندازد. با قرار دادن **throw()** پس از لیست پارامتری یک تابع، نشان داده می‌شود که تابع استثناهای را به جریان نخواهد انداخت. اگر تابع مبادرت به راه‌اندازی یک استثنا کند، تابع **unexpected** احضار می‌شود. در بخش ۷-۱۶ نشان داده خواهد شد که چگونه تابع **unexpected** می‌تواند با فراخوانی تابع **set_unexpected** بهینه‌سازی شود.

۷-۱۶ پردازش استثناهای غیرمنتظره (unexpected)

تابع **unexpected** تابع ثبت شده با تابع **set_unexpected** را فراخوانی می‌کند (تعریف شده در فایل سرآیند <**exception**>). اگر هیچ تابعی به این روش ثبت نشده باشد، تابع **terminate** بصورت پیش‌فرض فراخوانی می‌شود. حالت‌های که تابع **terminate** فراخوانی می شود عبارتند از:

۱- مکانیزم استثنا قادر به یافتن یک **catch** مطابق با استثنا رخ داده نباشد.

۲- نابود کننده‌ای مبادرت به راه‌اندازی یک استثنا در زمان باز کردن پشته کند.

۳- مبادرت به راه‌اندازی مجدد یک استثنا شود زمانیکه به استثنا جاری رسیدگی نشده است.

۴- فراخوانی تابع **unexpected** در حالت پیش‌فرض که تابع **terminate** را فراخوانی می‌کند.

تابع **set_terminate** می‌تواند تصریح کننده تابعی باشد که به هنگام فراخوانی **terminate** احضار می‌شود. در غیر اینصورت **terminate** مبادرت به فراخوانی **abort** می‌کند، که برنامه را بدون فراخوانی نابود کننده‌های شی‌های باقیمانده از کلاس ذخیره‌سازی استاتیک یا اتوماتیک خاتمه می‌دهد. اینکار می‌تواند سبب فقدان منابع شود چرا که برنامه نابهنگام خاتمه می‌پذیرد.

هر یک از توابع **set_terminate** و **set_unexpected** یک اشاره‌گر به آخرین تابع فراخوانی شده توسط **terminate** و **unexpected** برگشت می‌دهند (صفر، در اولین فراخوانی). این ویژگی به برنامه‌نویس امکان می‌دهد تا اشاره‌گر تابع را ذخیره کرده و از اینرو بتواند آنرا بعداً بازیابی کند.

توابع **set_terminate** و **set_unexpected** اشاره‌گرهای بعنوان آرگومان برای توابعی که نوع برگشتی آنها **void** بوده و آرگومانی ندارند، دریافت می‌کند.

اگر آخرین عمل تابع خاتمه دهنده تعریف شده از سوی برنامه‌نویس نتواند به برنامه خاتمه دهد، تابع **abort** فراخوانی شده و به اجرای برنامه خاتمه می‌دهد (البته پس از اجرای عبارات تابع خاتمه دهنده تعریف شده از سوی برنامه‌نویس).

۱۶-۸ باز کردن پشته

زمانیکه یک استثنا به راه می‌افتد اما در یک قلمرو خاص گرفتار نمی‌شود، پشته فراخوانی تابع باز می‌شود، و سعی می‌شود تا استثنا در بلوک **try...catch** خارجی گرفتار گردد. منظور از باز کردن پشته فراخوانی تابع این است که در تابعی که استثنا رخ داده و نتوانسته گرفتار شود، تمام متغیرهای محلی در آن تابع نابود شده، تابع خاتمه یافته و کنترل به عبارتی که در اصل آن تابع را فراخوانی کرده برگشت داده شود. اگر یک بلوک **try** آن عبارت را احاطه کرده باشد، مبادرت به گرفتن استثنا خواهد کرد. اگر بلوک **try** آن عبارت را احاطه نکرده باشد، مجدداً باز کردن پشته صورت خواهد گرفت. اگر هیچ رسیدگی کننده **catch** این استثنا را گرفتار نسازد، تابع **terminate** فراخوانی می‌شود تا برنامه را خاتمه دهد. برنامه شکل ۱۶-۴ به بررسی باز کردن پشته پرداخته است.

```
1  // Fig. 16.4: Fig16_04.cpp
2  // Demonstrating stack unwinding.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <stdexcept>
8  using std::runtime_error;
9
10 // function3 throws run-time error
11 void function3() throw ( runtime_error )
12 {
13    cout << "In function 3" << endl;
14
15    // no try block, stack unwinding occur, return control to function2
16    throw runtime_error( "runtime_error in function3" );
17 } // end function3
18
19 // function2 invokes function3
20 void function2() throw ( runtime_error )
21 {
22    cout << "function3 is called inside function2" << endl;
23    function3(); // stack unwinding occur, return control to function1
24 } // end function2
25
26 // function1 invokes function2
27 void function1() throw ( runtime_error )
28 {
29    cout << "function2 is called inside function1" << endl;
30    function2(); // stack unwinding occur, return control to main
31 } // end function1
32
33 // demonstrate stack unwinding
34 int main()
35 {
36    // invoke function1
37    try
38    {
39       cout << "function1 is called inside main" << endl;
```

```
40       function1(); // call function1 which throws runtime_error
41    } // end try
42    catch ( runtime_error &error ) // handle run-time error
43    {
44       cout << "Exception occurred: " << error.what() << endl;
45       cout << "Exception handled in main" << endl;
46    } // end catch
47
48    return 0;
49 } // end main
```

```
function1 is called inside main
function2 is called inside function1
function3 is called inside function2
In function 3
Exception occurred: runtime_error in function3
Exception handled in main
```

شکل ٤-١٦ | باز کردن پشته (stack unwind).

در **main**، بلوک **try** (خطوط 41-37) مبادرت به فراخوانی تابع **funcation1** می‌کند (خطوط 31-27). سپس تابع **function1**، تابع **function2** را فراخوانی می‌کند (خطوط 24-20)، که آن هم تابع **function3** را فراخوانی می‌کند (خطوط 17-11). خط 16 از **function3** یک شی **runtime_error** را راه‌اندازی می‌کند. با این وجود، چون هیچ بلوک **try** عبارت **throw** را احاطه نکرده است (در خط 16)، باز کردن پشته رخ می‌دهد، **function3** در خط 16 خاتمه می‌یابد، سپس کنترل به عبارتی در **function2** برگشت داده می‌شود که **function3** را فراخوانی کرده بود (یعنی خط 23). چون هیچ بلوک **try**، خط 23 را احاطه نکرده است، مجدداً باز کردن پشته رخ می‌دهد، **function2** خاتمه یافته (در خط 23) و کنترل به عبارتی در **function1** برگشت داده می‌شود که **function2** را فراخوانی کرده بود (یعنی خط 30). چون هیچ بلوک **try**، خط 30 را احاطه نکرده است، باز کردن پشته یکبار دیگر اتفاق می‌افتد، **function1** در خط 30 خاتمه یافته و کنترل به عبارتی در **main** باز می‌گردد که تابع **function1** را فراخوانی کرده بود (یعنی خط 40). بلوک try در خطوط 41-37 این عبارت را احاطه کرده است، از اینرو اولین رسیدگی کننده **catch** قرار گرفته پس از این بلوک **try** پیدا شده (خطوط 46-42) و استثنا را گرفتار و آنرا پردازش می‌کند. خط 44 از تابع **what** برای نمایش پیغام استثنا استفاده کرده است. بخاطر داشته باشید که تابع **what** یک تابع **virtual** (مجازی) از کلاس **exception** است که می‌تواند توسط یک کلاس مشتق شده به کنار گذاشته شود تا پیغام خطای مناسب برگشت داده شود.

۹-۱٦ سازنده‌ها، نابود کننده‌ها و رسیدگی به استثنا

ابتدا اجازه دهید تا به بررسی مسئله‌ی بپردازیم که قبلاً مطرح کردیم، اما به قدر کفایت راضی کننده نبود: زمانیکه یک خطا در یک سازنده تشخیص داده می‌شود چه اتفاقی می‌افتد؟ برای مثال، چگونه باید سازنده یک شی به هنگام رخ دادن واماندگی **new** از خود واکنش نشان دهد، چرا که سازنده قادر نیست تا حافظه مورد نیاز برای ذخیره‌سازی نمایندگی درونی آن شی را اخذ کند؟ بدلیل اینکه سازنده نمی‌تواند مقداری را

که نشاندهنده یک خطا است برگشت دهد، بایستی یک روش جایگزین انتخاب کنیم که نشان دهد شی بدرستی ایجاد نشده است. یک روش این است که شی که بدرستی ایجاد نشده است را برگشت داده و امیدوار باشیم که هر کسی که از آن استفاده می‌کند، تست یا آزمون‌های مناسبی بر روی آن اعمال کند تا مشخص شود آن شی در وضعیت پایداری قرار ندارد. روش دیگر تنظیم چند متغیر خارج از سازنده است.

شاید بهترین جایگزین این باشد که سازنده را ملزم به راه‌اندازی به یک استثنا کنیم که حاوی اطلاعات خطا باشد و از اینرو به برنامه فرصت مناسبی برای رسیدگی به واماندگی داده می‌شود. راه‌اندازی استثناها توسط یک سازنده سبب می‌شود نابود کننده‌ها برای هر شی که بعنوان بخشی از شی قبل از راه‌اندازی استثنا ایجاد شده‌اند، فراخوانی شوند. نابود کننده‌ها برای هر شی اتوماتیک ایجاد شده در هر بلوک **try** قبل از اینکه استثنا رخ داده باشد، فراخوانی می‌شود. باز کردن پشته تضمینی بر انجام کار از نقطه‌ای است که رسیدگی کننده به استثنا شروع به اجرا شدن می‌کند. اگر نابود کننده‌ای بعنوان نتیجه‌ای از باز کردن پشته فراخوانی گردد، تابع **terminate** فراخوانی خواهد شد.

اگر شی دارای شی‌های عضو باشد، و اگر یک استثنا قبل از اینکه شی خارجی کاملاً ایجاد شود، اتفاق افتد، پس نابود کننده‌ها برای شی‌های عضوی که قبل از رخ دادن استثنا ساخته شده‌اند، فراخوانی خواهند شد. اگر آرایه‌ای از شی‌های به هنگام رخ دادن یک استثنا تا حدی ایجاد شده باشد، نابود کننده‌ها فقط بر روی شی‌های ساخته شده در آرایه فراخوانی خواهند شد.

یک استثنا می‌تواند مانع از عملیات کدی شود که می‌خواهد بطرز عادی منبعی را رها سازد، از اینرو سبب، فقدان منبع می‌شود، یکی از تکنیک‌های حل این مشکل، مقداردهی اولیه یک شی محلی با آن منبع است. زمانیکه استثنا رخ می‌دهد، نابود کننده برای آن شی فراخوانی و می‌تواند منبع را آزاد کند.

۱۰-۱۶ استثناها و توارث

کلاس‌های استثنا گوناگونی را می‌توان از یک کلاس مبنای مشترک یا عمومی مشتق کرد، همانطوری که در بخش ۳-۱۶ زمانیکه کلاس **DivideByZeroException** را بعنوان یک کلاس مشتق شده از کلاس **exception** ایجاد کردیم. اگر یک رسیدگی کننده **catch** یک اشاره‌گر یا مراجعه‌ای را به یک شی استثنا از یک نوع کلاس مبنا، را گرفتار سازد، در ضمن می‌تواند یک اشاره‌گر یا مراجعه به تمام شی‌ها از کلاس‌های عمومی مشتق شده از آن کلاس مبنا را هم گرفتار کند. این فرآیند امکان پردازش چند ریختی خطاهای مرتبط را فراهم می‌آورد.

۱۱-۱۶ پردازش واماندگی new

زبان C++ استاندارد تصریح می‌کند که در زمان واماندگی عملگر **new**، استثنا **bad_alloc** به راه می‌افتد (تعریف شده در فایل سرآیند <**new**>). با این وجود، برخی از کامپایلرها سازگار با C++ استاندارد نیستند و بنابر این از آن نسخه new استفاده می‌کند که به هنگام واماندگی، صفر برگشت می‌دهد. برای مثال Microsoft Visual Studio .NET در زمان واماندگی **new** یک استثنا **bad_alloc** ایجاد می‌کند، در حالیکه Microsoft Visual C++ 6.0 در زمان واماندگی **new**، مقدار صفر برگشت می‌دهد.

کامپایلر در رسیدگی به واماندگی **new** به روش‌های متفاوتی عمل می‌کنند. بسیاری از کامپایلرهای قدیمی **C++** به هنگام واماندگی **new** بصورت پیش‌فرض صفر برگشت می‌دهند. برخی از کامپایلرها اگر فایل سرآیند <**new**> (یا <**new.h**>) بکار گرفته شده باشد، از راه‌اندازی یک استثنا توسط **new** پشتیبانی می‌کنند. کامپایلرهای دیگر در حالت پیش‌فرض **bad_alloc** را به راه می‌اندازند، صرفنظر از اینکه آیا فایل سرآیند <**new**> بکار گرفته شده است یا خیر. برای کسب اطلاعات بیشتر در این زمینه به مستندات کامپایلر خود مراجعه کنید.

در این بخش، به عرض سه مثال در ارتباط با واماندگی **new** می‌پردازیم. مثال اول در زمان واماندگی **new**، صفر برگشت می‌دهد. مثال دوم از نسخه‌ای از **new** استفاده می‌کند که در زمان واماندگی **new** یک استثنا **bad_alloc** به راه می‌اندازد. مثال سوم از تابع **set_new_handler** برای رسیدگی به واماندگی **new** استفاد می‌کند. [نکته: مثال‌های مطرح شده در شکل‌های ۵-۱۶ الی ۷-۱۶ میزان بسیاری زیادی از حافظه دینامیکی کامپیوتر را اخذ می‌کنند، از اینرو می‌توانند کامپیوتر شما را آهسته نمایند.]

برگشت دادن صفر در واماندگی new

برنامه شکل ۵-۱۶ به بررسی **new** پرداخته که در زمان واماندگی (یعنی زمانیکه تقاضای اخذ مقداری از حافظه را می‌کند و موفق نمی‌شود) صفر برگشت می‌دهد. عبارت **for** در خطوط 14-13 حلقه‌ای بوجود آورده که 50 بار تکرار می‌شود و در هر گذار، یک آرایهٔ 50.000.000 برای مقادیر از نوع **double** اخذ می‌کند (یعنی 400.000.000 بایت، چرا که یک double معمولاً 8 بایت است). عبارت **if** در خط 17 نتیجه هر عملیات new را تست می‌کند تا مشخص کند که آیا اخذ حافظه توسط **new** با موفقیت همراه بوده است یا خیر. اگر new دچار واماندگی شود و صفر برگشت دهد، خط 19 یک پیغام خطا چاپ کرده و حلقه خاتمه می‌یابد. [نکته: ما از Microsoft Visual C++ 6.0 در اجرای این مثال استفاده کرده‌ایم، چرا که Microsoft Visual Studio .NET در زمان واماندگی **new** بجای صفر، یک استثنا **bad_alloc** راه‌اندازی می‌کند.]

```
1  // Fig. 16.5: Fig16_05.cpp
2  // Demonstrating pre-standard new returning 0 when memory
3  // is not allocated.
```

```
4  #include <iostream>
5  using std::cerr;
6  using std::cout;
7
8  int main()
9  {
10    double *ptr[ 50 ];
11
12    // allocate memory for ptr
13    for ( int i = 0; i < 50; i++ )
14    {
15       ptr[ i ] = new double[ 50000000 ];
16
17       if ( ptr[ i ] == 0 ) // did new fail to allocate memory
18       {
19          cerr << "Memory allocation failed for ptr[ " << i << " ]\n";
20          break;
21       } // end if
22       else // successful memory allocation
23          cout << "Allocated 50000000 doubles in ptr[ " << i << " ]\n";
24    } // end for
25
26    return 0;
27 } // end main
```

```
Allocated 50000000 doubles in ptr[0]
Allocated 50000000 doubles in ptr[1]
Allocated 50000000 doubles in ptr[2]
Memory allocation failed for ptr[3]
```

شکل ٥-١٦ | برگشت صفر در زمان واماندگی new.

خروجی برنامه نشان می‌دهد که برنامه فقط سه بار قادر به تکرار حلقه قبل از واماندگی new بوده و حلقه خاتمه یافته است. براساس میزان حافظه فیزیکی، فضای دیسک در دسترس برای حافظه مجازی بر روی سیستم شما و کامپایلری که بکار گرفته‌اید، احتمالاً خروجی شما با خروجی نشان داده شده در اینجا متفاوت خواهد بود.

راه‌اندازی bad_alloc در زمان واماندگی new

برنامه شکل ۶-۱۶ نشان می‌دهد که در زمان واماندگی **new** در اخذ حافظه مورد نیاز، **bad_alloc** راه‌اندازی شده است. عبارت **for** در خطوط 24-20 در درون بلوک **try** بایستی 50 بار تکرار شود و در هر گذار، یک آرایه 50.000.000 برای مقادیر **double** اخذ شود، اگر **new** دچار واماندگی شود، یک استثنا **bad_alloc** به راه افتاده، حلقه خاتمه می‌یابد و برنامه از خط 28 بکار خود ادامه می‌دهد محلی که رسیدگی کننده **catch** استثنا را گرفتار کرده و آنرا پردازش می‌کند.

```
1  // Fig. 16.6: Fig16_06.cpp
2  // Demonstrating standard new throwing bad_alloc when memory
3  // cannot be allocated.
4  #include <iostream>
5  using std::cerr;
6  using std::cout;
7  using std::endl;
8
9  #include <new> // standard operator new
10 using std::bad_alloc;
11
12 int main()
13 {
14    double *ptr[ 50 ];
15
16    // allocate memory for ptr
```

```
17    try
18    {
19       // allocate memory for ptr[ i ]; new throws bad_alloc on failure
20       for ( int i = 0; i < 50; i++ )
21       {
22          ptr[ i ] = new double[ 50000000 ]; // may throw exception
23          cout << "Allocated 50000000 doubles in ptr[ " << i << " ]\n";
24       } // end for
25    } // end try
26
27    // handle bad_alloc exception
28    catch ( bad_alloc &memoryAllocationException )
29    {
30       cerr << "Exception occurred: "
31          << memoryAllocationException.what() << endl;
32    } // end catch
33
34    return 0;
35 } // end main
```

```
Allocated 50000000 doubles in ptr[0]
Allocated 50000000 doubles in ptr[1]
Allocated 50000000 doubles in ptr[2]
Exception occurred: bad allocation
```

شکل ۶-۱۶ | برگشت bad_alloc در زمان واماندگی new.

رسیدگی کننده **catch** استثنا را گرفته و پردازش می‌کند. خطوط 31-30 پیغام "Exception occurred:" را بدنبال پیغام برگشتی از تابع **what** چاپ می‌کنند. خروجی نشان می‌دهد که برنامه فقط سه بار حلقه را قبل از واماندگی **new** و راه افتادن استثنا **bad_alloc** تکرار کرده است. ممکن خروجی برنامه بر روی کامپیوتر شما با خروجی این برنامه متفاوت باشد.

زبان C++ استاندارد تصریح می‌کند که کامپایلرهای استاندارد سازگار می‌تواند به استفاده از نسخه‌ای از **new** که در مواجه با واماندگی صفر برگشت می‌دهند، ادامه دهند. به همین منظور، فایل سرآیند <**new**> شی **nothrow** (از نوع **nothrow_t**) را تعریف کرده است که بصورت زیر بکار گرفته می‌شود:

```
double *ptr = new( nothrow ) double[50000000];
```

عبارت فوق از نسخه‌ای از **new** استفاده می‌کند که به هنگام اخذ حافظه برای آرایه 50.000.000 از نوع **double** استثنا **bad_alloc** را سبب نمی‌شود (یعنی **nothrow**)..

رسیدگی به واماندگی new با استفاده از تابع set_new_handler

یکی از ویژگیهای دیگر در رسیدگی به واماندگی **new** تابع **set_new_handler** است (نمونه اولیه در فایل سرآیند استاندارد <**new**>). این تابع یک اشاره‌گر به تابعی که هیچ آرگومانی دریافت نمی‌کند و **void** برگشت می‌دهد، بعنوان آرگومان دریافت می‌کند. این اشاره‌گر به تابعی اشاره دارد که اگر **new** دچار واماندگی شود، فراخوانی خواهد شد. این قابلیت یک روش منسجم در رسیدگی به تمام واماندگی‌های **new**، صرفنظر از اینکه واماندگی در کجای برنامه رخ داده است، در اختیار برنامه‌نویس قرار می‌دهد. زمانیکه **set_new_handler** یک رسیدگی کننده **new** در برنامه را ثبت کرد، عملگر **new** نمی‌تواند در زمان واماندگی مبادرت به راه‌اندازی **bad_alloc** کند و بجای آن، خطا را تحویل تابع رسیدگی کننده **new** می‌دهد.

اگر **new** موفق شود، حافظه مورد نیاز را اخذ کند، یک اشاره‌گر به آن حافظه برگشت خواهد داد. اگر **new** در اخذ حافظه دچار واماندگی شود و **set_new_handler** برای یک تابع رسیدگی کننده **new** ثبت نشده باشد، آنگاه یک استثنا **bad_alloc** به جریان خواهد انداخت. اگر **new** در اخذ حافظه دچار واماندگی شود و تابع رسیدگی کننده **new** ثبت شده باشد، این تابع فراخوانی خواهد شد. C++ استاندارد تصریح می‌کند که تابع رسیدگی کننده **new** بایستی یکی از وظایف زیر را انجام دهد:

۱- تهیه حافظه مورد نیاز با حذف سایر حافظه‌های اخذ شده دینامیکی (یا به کاربر اعلان شود تا برنامه‌های دیگر را خاتمه دهد) و برگشت به عملگر **new** برای مبادرت به اخذ مجدد حافظه.

۲- راه‌اندازی یک استثنا از نوع **bad_alloc**.

۳- فراخوانی تابع **abort** یا **exit** (که هر دو در فایل سرآیند <**cstdlib**> وجود دارند) برای خاتمه دادن برنامه.

برنامه شکل ۷-۶ به بررسی **set_new_handler** پرداخته است. تابع **customNewHandler** در خطوط 14-18 یک پیغام خطا چاپ کرده (خط 16)، سپس برنامه را از طریق فراخوانی **abort** خاتمه می‌دهد (خط 17). خروجی نشان می‌دهد که برنامه فقط سه بار قبل از واماندگی **new** و فراخوانی تابع **customNewHandler** موفق به تکرار حلقه شده است. مجدداً احتمال دارد خروجی این برنامه بر روی کامپیوتر شما با خروجی به نمایش در آمده در اینجا متفاوت باشد.

```
1  // Fig. 16.7: Fig16_07.cpp
2  // Demonstrating set_new_handler.
3  #include <iostream>
4  using std::cerr;
5  using std::cout;
6
7  #include <new> // standard operator new and set_new_handler
8  using std::set_new_handler;
9
10 #include <cstdlib> // abort function prototype
11 using std::abort;
12
13 // handle memory allocation failure
14 void customNewHandler()
15 {
16    cerr << "customNewHandler was called";
17    abort();
18 } // end function customNewHandler
19
20 // using set_new_handler to handle failed memory allocation
21 int main()
22 {
23    double *ptr[ 50 ];
24
25    // specify that customNewHandler should be called on
26    // memory allocation failure
27    set_new_handler( customNewHandler );
28
29    // allocate memory for ptr[ i ]; customNewHandler will be
30    // called on failed memory allocation
31    for ( int i = 0; i < 50; i++ )
32    {
33       ptr[ i ] = new double[ 50000000 ]; // may throw exception
```

```
34       cout << "Allocated 50000000 doubles in ptr[ " << i << " ]\n";
35    } // end for
36
37    return 0;
38 }  // end main
```

```
Allocated 50000000 doubles in ptr[0]
Allocated 50000000 doubles in ptr[1]
Allocated 50000000 doubles in ptr[2]
customNewHandler was called
```

شکل ۷-۱۶ | set_new_handler نشاندهنده فراخوانی تابع در زمان واماندگی new است.

۱۲-۱۶ کلاس auto_ptr و تخصیص حافظه دینامیکی

یکی از رویه‌های معمول در برنامه‌نویسی اخذ یا تخصیص حافظه دینامیکی، تخصیص آدرس آن حافظه به یک اشاره‌گر، استفاده از اشاره‌گر برای دستکاری کردن حافظه و رهاسازی حافظه با **delete** در زمانی است که دیگر به آن حافظه نیازی نداریم. اگر یک استثنا پس از تخصیص موفقیت‌آمیز حافظه رخ دهد اما قبل از اجرای عبارت **delete** باشد، فقدان حافظه می‌تواند اتفاق بیفتد. زبان C++ استاندارد الگوی کلاس **auto_ptr** را در سرآیند فایل <**memory**> تدارک دیده است که به این وضعیت رسیدگی می‌کند.

یک شی از کلاس **auto_ptr** یک اشاره‌گر به حافظه اخذ شده دینامیکی را نگهداری می‌کند. زمانیکه نابود کننده شی **auto_ptr** فراخوانی می‌شود (برای مثال، زمانیکه شی **auto_ptr** از قلمرو خارج می‌شود)، یک عملیات **delete** بر روی اشاره‌گر عضو داده خود انجام می‌دهد. الگوی کلاس **auto_ptr** سربارگذاری عملگرهای * و -> را تدارک دیده است و از اینروست که یک شی **auto_ptr** می‌تواند بعنوان یک متغیر اشاره‌گر عادی بکار گرفته شود. برنامه شکل۱۰-۱۶ به بررسی یک شی **auto_ptr** پرداخته است که به یک شی اخذ شده دینامیکی از کلاس **Integer** اشاره دارد (شکل۸-۱۶ و ۹-۱۶).

خط 18 از شکل ۱۰-۱۶ شی **ptrToInteger** را از **auto_ptr** ایجاد کرده و آنرا با یک اشاره‌گر به شی **Integer** اخذ شده دینامیکی مقداردهی می‌کند که حاوی مقدار 7 است. خط 21 از عملگر سربارگذاری شده -> برای احضار تابع **setInteger** بر روی شی **Integer** مورد اشاره توسط **ptrToInteger** استفاده کرده است. خط 24 از عملگر سربارگذاری شده * برای بازیابی اطلاعات از طریق **ptrToInteger** استفاده کرده است، سپس از عملگر نقطه (.) برای فراخوانی تابع **getInteger** بر روی شی **Integer** مورد اشاره توسط **ptrToInteger** سود برده است همانند یک اشاره‌گر عادی، عملگرهای سربارگذاری شده -> و * می‌توانند در دسترسی به شی که **auto_ptr** به آن اشاره می‌کند، بکار گرفته شوند.

بدلیل اینکه **ptrToInteger** یک متغیر اتوماتیک محلی به **main** است، **ptrToInteger** در زمان خاتمه **main** نابود می‌شود. نابود کننده **auto_ptr** یک **delete** را بر روی شی **Integer** اشاره شده توسط **ptrToInteger** اعمال می‌کند، که در ادامه نابود کننده کلاس **Integer** فراخوانی می‌شود. حافظه‌ای که **Integer** اشغال کرده بود آزاد می‌شود، صرفنظر از اینکه چگونه کنترل از بلوک خارج شده باشد (یعنی

توسط یک عبارت **return** یا توسط یک استثنا). از همه مهمتر، با استفاده از یک تکنیک می‌توان جلوی فقدان حافظه را گرفت.

```
1  // Fig. 16.8: Integer.h
2  // Integer class definition.
3
4  class Integer
5  {
6  public:
7     Integer( int i = 0 ); // Integer default constructor
8     ~Integer(); // Integer destructor
9     void setInteger( int i ); // set Integer value
10    int getInteger() const; // return Integer value
11 private:
12    int value;
13 }; // end class Integer
```

شکل ۸-۱٦ | تعریف کلاس Integer.

```
1  // Fig. 16.9: Integer.cpp
2  // Integer member function definition.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Integer.h"
8
9  // Integer default constructor
10 Integer::Integer( int i )
11    : value( i )
12 {
13    cout << "Constructor for Integer " << value << endl;
14 } // end Integer constructor
15
16 // Integer destructor
17 Integer::~Integer()
18 {
19    cout << "Destructor for Integer " << value << endl;
20 } // end Integer destructor
21
22 // set Integer value
23 void Integer::setInteger( int i )
24 {
25    value = i;
26 } // end function setInteger
27
28 // return Integer value
29 int Integer::getInteger() const
30 {
31    return value;
32 } // end function getInteger
```

شکل ۹-۱٦ | تعریف تابع عضو از کلاس Integer.

```
1  // Fig. 16.10: Fig16_10.cpp
2  // Demonstrating auto_ptr.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <memory>
8  using std::auto_ptr; // auto_ptr class definition
9
10 #include "Integer.h"
11
12 // use auto_ptr to manipulate Integer object
13 int main()
14 {
15    cout << "Creating an auto_ptr object that points to an Integer\n";
16
```

```
17    // "aim" auto_ptr at Integer object
18    auto_ptr< Integer > ptrToInteger( new Integer( 7 ) );
19
20    cout << "\nUsing the auto_ptr to manipulate the Integer\n";
21    ptrToInteger->setInteger( 99 ); // use auto_ptr to set Integer value
22
23    // use auto_ptr to get Integer value
24    cout << "Integer after setInteger: "<<( *ptrToInteger ).getInteger()
25       << "\n\nTerminating program" << endl;
26    return 0;
27 }  // end main
```

```
Creating an auto_ptr object that points to an Integer
Constructor for Integer 7

Using the auto_ptr to manipulate the Integer
Integer after setInteger: 99

Terminating program
Destructor for Integer 99
```

شکل ۱۰-۱۶ | شی auto_ptr تخصیص حافظه دینامیکی را مدیریت می‌کند.

یک **auto_ptr** می‌تواند مالکیت حافظه دینامیکی را که مدیریت می‌کند از طریق عملگر تخصیص سربارگذاری شده خود یا سازنده کپی کننده انتقال دهد. آخرین شی **auto_ptr** که اشاره‌گر به حافظه دینامیکی را نگهداری می‌کند، حافظه را حذف خواهد کرد. این ویژگی **auto_ptr** را تبدیل به مکانیزم مناسبی را برای باز گرداندن حافظه اخذ شده دینامیکی به کد سرویس‌گیرنده کرده است. زمانیکه **auto_ptr** در کد سرویس‌گیرنده از قلمرو خارج می‌شود، نابود کننده **auto_ptr** حافظه دینامیکی را حذف می‌کند.

۱۳-۱۶ سلسله مراتب استاندارد استثنا

تجربه نشان داده است که استثناها بخوبی در یکی از چندین رده تعیین شده قرار می‌گیرند. کتابخانه استاندارد C++ شامل یک سلسله مراتب از کلاس‌هایی استثنا است (شکل ۱۱-۱۶). همانطوری که در ابتدای بخش ۳-۱۶ بیان کردیم، سرسلسله این سلسله مراتب کلاس مبنای **exception** است (تعریف شده در فایل سرآیند <**exception**>)، که حاوی تابع مجازی **what** است که کلاس‌های مشتق شده می‌توانند پیغام‌های مناسب خطا را توسط آن صادر کنند.

بلافاصله پس از کلاس مبنای **exception** کلاس‌های مشتق شده **runtime_error** و **logic_error** قرار دارند (هر دو در سرآیند <**stdexcept**> تعریف شده‌اند)، هر یک از این دو، دارای چندین کلاس مشتق شده هستند. همچنین از کلاس **exception** استثناهای مشتق شده‌اند که توسط عملگرهای C++ به راه می‌افتند، برای مثال، **bad_alloc** توسط **new** (بخش ۱۱-۶)، **bad_cast** توسط **dynamic_cast** (فصل ۱۳) و **bad_typeid** توسط **typeid** (فصل ۱۳) به جریان می‌افتند. منظور از **bad_exception** این است که اگر یک استثنا غیرمنتظره رخ دهد، تابع **unexpected** می‌تواند **bad_exception** را بجای خاتمه دادن به

اجرای برنامه (حالت پیش‌فرض) با فراخوانی تابع دیگر مشخص شده توسط **set_unexpected** راه‌اندازی کند.

شکل ۱۱-۱۶ | کلاس‌های استثنا در کتابخانه استاندارد.

کلاس **logic_error** کلاس مبنا برای چندین کلاس استثنا است که دلالت بر خطا در منطق برنامه دارند. برای مثال، کلاس **invalid_argument** بر این نکته دلالت دارد که یک آرگومان نامعتبر به تابع ارسال شده است. کلاس **length_error** نشان می‌دهد که طول، بیش از حداکثر سایز اجازه داده شده برای آن شی می‌باشد. کلاس **out_of_range** نشان می‌دهد که مقداری همانند شاخص یک آرایه از مرزهای آرایه تجاوز کرده است.

کلاس **runtime_error** که بطور خلاصه در بخش ۸-۱۶ بکار گرفته شد، کلاس مبنا برای چند کلاس استثنا استاندارد است که دلالت بر خطاهای زمان اجرا دارند. برای مثال، کلاس **overflow_error** نشاندهنده خطای سرریز محاسباتی و کلاس‌های **underflow_error** دلالت بر خطای پاریز دارد (یعنی نتیجه یک عملیات عددی کوچکتر از کوچکترین عددی است که می‌توان در کامپیوتر ذخیره کرد).

۱۴-۱۶ تکنیک‌های رسیدگی به خطا

در سرتاسر این فصل به بیان انواع روش‌های مقابله با استثناهای رخ داده پرداختیم. در این بخش بصورت چکیده این تکنیک‌ها و سایر تکنیک‌های مرتبط با رسیدگی به خطا را بیان می‌کنیم:

- نادیده گرفتن استثنا. اگر استثنای رخ دهد، برنامه می‌تواند در مقابل استثنا گرفتار نشده دچار واماندگی شود. این حالت می‌تواند برای محصولات نرم‌افزاری تجاری یا نرم‌افزارهای خاصی که از اهمیت خاص و حیاتی برخوردار هستند، عیب محسوب می‌شود، اما برای نرم‌افزارهای که برای خودتان طراحی می‌کنید، می‌توان برخی از خطاها را نادیده گرفت.
- خاتمه برنامه. البته اینکار جلوی اجرای برنامه را گرفته و از تولید نتایج اشتباه ممانعت بعمل می‌آورد. برای بسیاری از انواع خطاها، این روش مناسب است، بویژه برای خطاهای غیرعظیم **(nonfatal)** که به برنامه اجازه می‌دهند تا اجرای خود را دنبال کند (در صورتیکه برنامه با خطا کار خود را دنبال می‌کند). این استراتژی برای برنامه‌های کاربردی حیاتی مناسب نمی‌باشد. البته بحث منابع در اینجا مهم است. اگر برنامه یک منبع را بدست گرفته باشد، بایستی قبل از اینکه برنامه خاتمه یابد، منبع را رها کند.
- تنظیم شاخص‌های خطا. مشکلی که این روش دارد این است که برنامه نمی‌تواند در تمام وضعیت‌ها مبادرت به تنظیم این شاخص‌های خطا کند.

• تست کردن شرط خطا، ارسال پیغام خطا و فراخوانی **exit** (در <**cstdlib**>) برای ارسال کد خطا متناسب به محیط برنامه.

• استفاده از توابع **setjump** و **longjump**. این توابع کتابخانه‌ای از <**csetjmp**> به برنامه‌نویس امکان می‌دهند تا یک پرش بلادرنگ از عمق یک فراخوانی تودرتو تابع به یک رسیدگی کننده خطا، انجام دهد. بدون استفاده از **setjump** یا **longjump**، برنامه باید چندین برگشت انجام دهد تا از عمق فراخوانی تودرتوی تابع خارج گردد. توابع **setjump** و **longjump** توابع خطرناکی هستند، چرا که مبادرت به باز کردن پشته می‌کنند بدون اینکه نابود کننده‌ها برای شی‌های اتوماتیک فراخوانی شوند. خود همین مسئله می‌تواند مشکلات جدی بدنبال داشته باشد.

• برخی از انواع خطاهای خاص دارای قابلیت‌های اختصاصی در رسیدگی به مشکل هستند. برای مثال، زمانیکه عملگر **new** دچار واماندگی می‌شود، می‌تواند تابع **new_handler** را برای رسیدگی به خطا به اجرا در آورد.

فصل
هفدهم

پردازش فایل

اهداف

- **ایجاد، خواندن، نوشتن و به روز کردن فایل‌ها.**
- **پردازش فایل‌های ترتیبی.**
- **پردازش فایل‌ها با دسترسی تصادفی.**
- **استفاده از عملیات I/O قالب‌بندی نشده با کارایی بالا.**
- **ایجاد یک برنامه تراکنشی با استفاده از پردازش فایل با دسترسی تصادفی.**

رئوس مطالب

۱۷-۱ مقدمه

متغیرها و آرایه‌ها، فقط قادر به نگهداری موقت داده‌ها هستند. زمانیکه یک متغیر محلی به خارج از قلمرو خود می‌رود یا هنگامی که برنامه خاتمه می‌یابد، داده‌ها از بین می‌روند. در مقابل، از فایل‌ها برای نگهداری طولانی مدت حجم زیادی از اطلاعات، حتی در زمانیکه برنامه ایجاد کننده آنها خاتمه می‌پذیرد، استفاده می‌شود. کامپیوترها، فایل‌ها را بر روی دستگاههای ذخیره‌سازی ثانویه، همانند دیسک‌های مغناطیسی، دیسک‌های نوری و نوارهای مغناطیسی ذخیره می‌کنند. در این فصل، به بررسی نحوه ایجاد، به روز کردن و پردازش داده‌های فایل‌ها در برنامه‌های C++ می‌پردازیم. در مورد هر دو نوع نحوهٔ دسترسی به فایل یعنی ترتیبی و تصادفی صحبت خواهیم کرد. یکی از قابلیت‌های بسیار مهم در هر زبان برنامه‌نویسی، پردازش فایل است چرا که با وجود این توانایی، می‌توان برنامه‌های تجاری ایجاد کرد. چنین برنامه‌های می‌توانند حجم زیادی از اطلاعات را پردازش کنند.

۲-۱۷ سلسله مراتب داده

عاقبت تمام ایتم‌های داده توسط کامپیوتر به ترکیب‌هایی از صفرها و یک‌ها تبدیل می‌شوند. دلیل این امر ساده و اقتصادی بودن ساخت قطعات الکترونیکی است که براساس دو وضعیت پایدار یعنی 0 و 1 کار می‌کنند.

کوچکترین ایتم داده که کامپیوترها از آن پشتیبانی می‌کنند، *بیت* نامیده می‌شود (کوتاه شده عبارت "*binary digit*" یا رقم باینری است، یک رقم می‌تواند یکی از دو مقدار صفر یا یک باشد). هر ایتم داده یا بیت، می‌تواند بعنوان مقدار صفر یا یک فرض گردد. مدارات کامپیوتر اعمال ساده‌ای بر روی بیت‌ها انجام می‌دهند، اعمالی مانند بررسی مقدار یک بیت، تنظیم مقدار بیت و معکوس کردن بیت (از **1** به **0** یا از **0** به **1**).

اصولاً برنامه‌نویس ترجیح می‌دهد که با این سطح از داده‌ها کار نکند و بجای آن با داده‌هایی مانند ارقام دهدهی (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) حروف (از A تا Z و a تا z)، نمادهای ویژه (همانند $، @، %، &، *، (،)، -، +، "، :، ?، /) کار کند. به ارقام، حروف و نمادهای ویژه *کاراکتر* گفته می‌شود. از مجموعه این کاراکترها برای نوشتن برنامه‌ها استفاده می‌شود و مجموعه این کاراکترها در یک کامپیوتر *مجموعه کاراکترهای* آن کامپیوتر نامیده می‌شوند. بدلیل اینکه کامپیوترها می‌توانند فقط با **1**ها و **0**ها کار کنند، هر کاراکتری در مجموعه کاراکتری یک کامپیوتر با استفاده از تعدادی 1 و 0 ارائه می‌شود که ترکیب آنها با یکدیگر یک یا چند *بایت* را تشکیل می‌دهد. بایت‌ها از ترکیب (مجموع) هشت بیت ساخته می‌شوند. برنامه‌نویس برنامه و داده‌ها را با استفاده از کاراکترها ایجاد می‌کند. کاراکترها ترکیبی از بیت‌ها هستند و ترکیبی از کاراکترها یا بایت‌ها، فیلدها را بوجود می‌آورند. یک فیلد گروهی از کاراکترهاست که معنی و مفهوم خاصی ایجاد می‌کند. برای مثال، یک فیلد شامل حروف بزرگ و کوچک می‌تواند در نشان دادن نام شخصی مورد استفاده قرار گیرد.

عناصر داده توسط کامپیوتر از طریق سلسله مراتب داده (شکل ۱-۱۷)، که این عناصر را به ساختار بزرگ و پیچیده‌ای که آنرا از بیت‌ها ساخته‌ایم، به کاراکترها تبدیل می‌کند.

بطور کلی، یک *رکورد* ترکیبی از چند فیلد است. برای مثال، در یک سیستم پرداخت حقوق، یک رکورد برای کارمند ممکن است دارای فیلدهای زیر باشد:

۱- شمارهٔ شناسایی کارمند

۲- نام

۳- آدرس

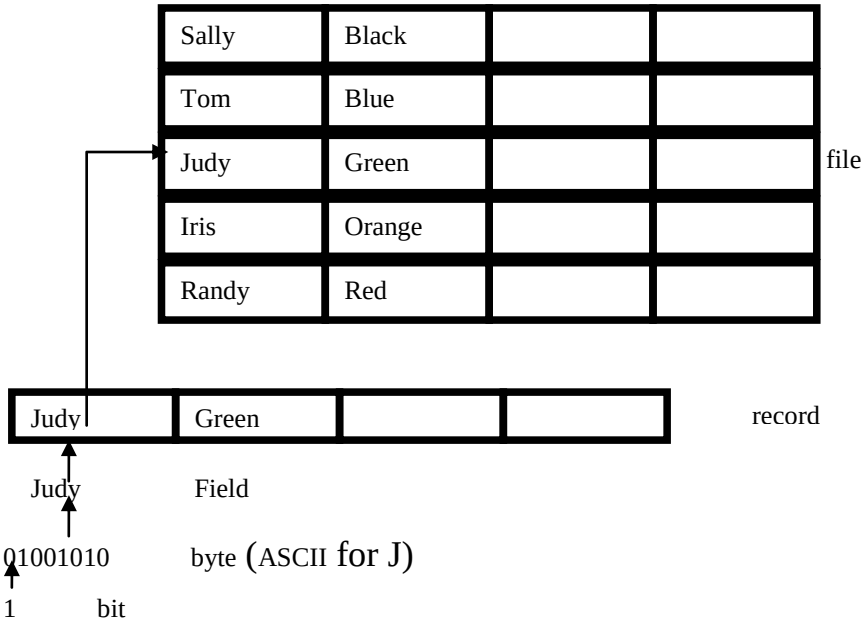
۴- نرخ دستمزد ساعتی

۵- تعداد مرخصی

۶- سال استخدام

۷- مقدار مالیات بر درآمد

بنابراین یک رکورد گروهی از فیلدهای مرتبط باهم است. در مثالی که آورده شدهٔ هر کدام یک از این فیلدها متعلق به یک کارمند است. البته یک شرکت ممکن است تعداد زیادی کارمند داشته باشد و در اینحالت هر کارمند رکورد مخصوص خود را خواهد داشت. یک فایل گروهی از رکوردهای مرتبط با یکدیگر است. یک فایل دستمزد بطور عادی شامل یک رکورد برای هر کارمند می‌باشد. برای یک شرکت کوچک فایل دستمزد ممکن است فقط 22 رکورد داشته باشد، در حالیکه یک شرکت بزرگ ممکن است بیشتر از صد هزار رکورد در فایل دستمزد خود داشته باشد.



شکل ۱-۱۷ | سلسله مراتب داده.

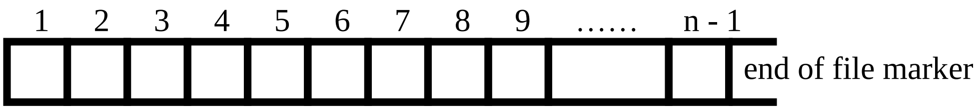
برای آسانتر کردن دستیابی به رکوردهای موجود در یک فایل یکی از فیلدهای هر رکورد بعنوان *کلید* رکورد انتخاب می‌شود. کلید رکورد، شناسه یک رکورد است که متعلق به یک شخص یا موجودیت می‌باشد و آن رکورد را از تمام رکوردهای دیگر متمایز می‌کند. در رکورد دستمزد، شماره شناسایی کارمند می‌تواند بعنوان کلید رکورد بکار گرفته شود. روش‌های متعددی برای سازماندهی رکوردها در

یک فایل وجود دارد. یکی از عمومی‌ترین نوع سازماندهی، *فایل ترتیبی* نامیده می‌شود. در این نوع از سازماندهی رکوردها به ترتیب فیلد کلید رکورد ذخیره می‌شوند. در فایل دستمزد، رکوردها به ترتیب شماره شناسایی کارمند در فایل قرار می‌گیرند. اولین رکورد در این فایل کوچکترین شماره‌شناسایی را خواهد داشت و به همین ترتیب رکوردهای بعدی دارای شماره‌شناسایی بالاتری خواهند بود.

بیشتر سازمان‌های تجاری از فایل‌های متفاوتی برای ذخیره داده‌ها استفاده می‌کنند. برای مثال یک کمپانی ممکن است دارای فایل‌های دستمزد، حقوق، درآمد، مشتری و غیره باشد. ارتباط چندین فایل با یکدیگر *پایگاه داده* (*Database*) نامیده می‌شود. به مجموعه برنامه‌های طراحی، ایجاد و مدیریت پایگاه داده، *سیستم مدیریت پایگاه داده* (DBMS[1]) اطلاق می‌شود.

۳-۱۷ فایل‌ها و استریم‌ها

نگاه ++C به هر فایل بفرم یک *استریم* (*stream*) متوالی از بایت‌ها است (شکل ۲-۱۷). انتهای هر فایل با یک نماد پایان فایل یا به تعداد بایت‌های مشخص شده که در سیستم مدیریت نگهداری ساختار داده به ثبت رسیده، تعیین می‌شود. هنگامی که یک فایل باز می‌شود،++C یک شیی ایجاد کرده و سپس آنرا به یک استریم مرتبط می‌کند. در فصل ۱۵ مشاهده کردید که شی‌های cin, cout, cerr و clog به هنگام استفاده از <iostram> ایجاد می‌شوند. این شیی‌ها ارتباط مابین یک برنامه و یک فایل مشخص یا دستگاه را تسهیل می‌بخشند. برای مثال شی **cin** به برنامه امکان می‌دهد تا داده را از طریق صفحه کلید بعنوان ورودی بپذیرد. شی **cout** به برنامه امکان می‌دهد تا داده به روی صفحه نمایش منتقل شود (خروجی). شی‌های **cerr** و **clog** به برنامه اجازه می‌دهند تا پیغام خطا را بر روی صفحه نمایش به نمایش در آورد.



شکل ۲-۱۷ | نگاه ++C به یک فایل n بایت‌ی.

برای پردازش فایل در ++C، لازم است تا فایل‌های سرآیند **<iostream>** و **<fstream>** بکار گرفته شوند. سرآیند **<fstream>** شامل تعاریفی برای الگوهای استریم کلاس **basic_ifstream** (برای فایل ورودی)، **basic_ofstream** (برای فایل خروجی) و **basic_fstream** (برای فایل ورودی و خروجی) است. هر الگوی کلاس دارای یک الگوی تخصصی از پیش تعریف شده است که **char I/O** را امکان‌پذیر می‌سازد. علاوه بر این، کتابخانه **fstream** مجموعه‌ای از *typedefها* تدارک دیده است که برای این الگوهای تخصصی اسامی مستعار تهیه می‌کنند. برای مثال **typedef ifstream** نشاندهنده یک

۱- Database Management System

basic_ifstream تخصصی شده است که ورودی **char** از یک فایل را فراهم می‌آورد. به همین ترتیب **typedef ofstream** نشاندهنده یک **basic_ofstream** تخصصی است که خروجی **char** به فایل‌ها را فراهم می‌آورد.

فایل‌ها با ایجاد شی‌ها از این الگوهای تخصصی استریم باز می‌شوند. این الگوها از الگوهای کلاس **basic_istream**، **basic_ostream** و **basic_iostream** مشتق می‌شوند. از اینرو، تمام توابع عضو، عملگرها و دستکاری کنندهایی که متعلق به این الگوها هستند نیز می‌تواند در استریم‌های فایل بکار گرفته شوند. شکل ۳-۱۷ بطور خلاصه رابطه توارث کلاس I/O را که تا بدین جا مطرح کرده‌ایم را نشان می‌دهد.

شکل ۳-۱۷ | بخشی از استریم سلسله مراتب الگوی I/O.

۱۷-٤ ایجاد فایل ترتیبی

C++ ساختاری بر روی فایل تحمیل نمی‌کند. از اینرو، مفاهیمی همانند "رکورد" در فایل‌های C++ وجود ندارد. به این معنی که، برنامه‌نویس بایستی ساختار فایل را به نحوی تدارک ببیند که نیاز برنامه را جوابگو باشد. در مثال بعدی، از کاراکترهای متنی و ویژه، برای سازماندهی مفهوم خاصی که برای "رکورد" قائل هستیم استفاده خواهیم کرد.

برنامه شکل ۴-۱۷ یک فایل ترتیبی ایجاد می‌کند که می‌تواند در یک سیستم دریافت کننده حساب به منظور مدیریت پول بکار گرفته شود. برای هر مشتری، برنامه یک شماره حساب، نام، نام‌خانوادگی و موجودی را فراهم می‌آورد. اطلاعات دریافتی برای هر مشتری، یک رکورد برای آن مشتری تشکیل می‌دهند. در این برنامه، شماره حساب، نشاندهنده کلید رکورد است. ایجاد و دستکاری کردن فایل‌ها براساس ترتیب شماره حساب صورت می‌گیرد. برنامه بر این فرض کار می‌کند که کاربر، براساس ترتیب شماره حساب رکوردها را وارد می‌کند. با این همه، یک سیستم کارآمد باید دارای قابلیت مرتب‌سازی نیز باشد. کاربر می‌تواند با هر ترتیبی، رکوردها را وارد کرده و سپس رکوردها مرتب شده و بصورت منظم در فایل نوشته شوند.

```
1  // Fig. 17.4: Fig17_04.cpp
2  // Create a sequential file.
3  #include <iostream>
4  using std::cerr;
5  using std::cin;
6  using std::cout;
7  using std::endl;
8  using std::ios;
9
10 #include <fstream> // file stream
11 using std::ofstream; // output file stream
12
13 #include <cstdlib>
14 using std::exit; // exit function prototype
15
```

```
16 int main()
17 {
18    // ofstream constructor opens file
19    ofstream outClientFile( "clients.dat", ios::out );
20
21    // exit program if unable to create file
22    if ( !outClientFile ) // overloaded ! operator
23    {
24       cerr << "File could not be opened" << endl;
25       exit( 1 );
26    } // end if
27
28    cout << "Enter the account, name, and balance." << endl
29       << "Enter end-of-file to end input.\n? ";
30
31    int account;
32    char name[ 30 ];
33    double balance;
34
35    // read account, name and balance from cin, then place in file
36    while ( cin >> account >> name >> balance )
37    {
38       outClientFile <<account <<' ' <<name <<' '<< balance << endl;
39       cout << "? ";
40    } // end while
41
42    return 0; // ofstream destructor closes file
43 } // end main
```

```
Enter the account, name, and balance.
Enter end-of-file to end input.
? 100 Jones 24.98
? 200 Doe 345.67
? 300 White 0.00
? 400 Stone -42.16
? 500 Rich 224.62
? ^Z
```

شکل ٤-١٧ | ایجاد فایل ترتیبی.

اجازه دهید تا به بررسی این برنامه بپردازیم. همانطوری که قبلاً گفته شد، فایلها با ایجاد شیءها **ifstream**، **ofstream** یا **fstream** باز می‌شوند. در شکل ۴-۱۷، فایل برای خروجی باز شده است، از اینرو یک شی **ofstream** ایجاد شده است. دو آرگومان به سازنده شی ارسال شده است، نام فایل و مد باز کردن فایل (خط 19). برای یک شی **ofsteam**، مد باز کردن فایل می‌تواند **ios::out** برای خارج کردن داده به یک فایل یا **ios::app** برای الحاق داده به انتهای فایل باشد (بدون اینکه تغییری در داده‌های حاضر در فایل اعمال کند). باز کردن فایل‌های موجود در مد **iso::out** سبب بریده شدن فایل می‌شود، به این معنی که تمام داده‌های موجود در فایل از بین می‌روند. اگر فایل از قبل وجود نداشته باشد، پس **ofstream** فایل را با استفاده از نام فایل ایجاد می‌کند.

خط 19 یک شی **ofstream** بنام **outClientFile** مرتبط با فایل **clients.dat** ایجاد می‌کند که برای خروجی باز شده است. آرگومان‌های **"clients.dat"** و **ios::out** به سازنده **ofstream** ارسال می‌شوند که فایل را باز کند. اینکار یک "خط ارتباطی" با فایل بنا می‌کند.

بطور پیش‌فرض شی‌های **ofstream** برای خروجی باز می‌شوند، از اینرو خط 19 می‌توانست عبارت زیر را به اجرا در آورد

```
ofstream outClientFile( "clients.dat" );
```

تا **clients.dat** را برای خروجی باز نماید. جدول ۵-۱۷ مدل‌های باز کردن فایل را لیست کرده است.

مد	توضیح
ios::app	تمام خروجی را به انتهای فایل الصاق می‌کند.
ios::ate	یک فایل برای خروجی باز کرده و به انتهای فایل حرکت می‌کند (معمولاً برای الصاق داده به فایل بکار گرفته می‌شود). داده می‌تواند در هر کجای فایل نوشته شود.
ios::in	فایل را برای ورودی باز می‌کند.
ios::out	فایل را برای خروجی باز می‌کند.
ios::trunc	اگر فایل حاوی اطلاعات باشد، آنها را از بین می‌برد (پیش‌فرض **ios:out** است).
ios:binary	فایل را برای باینری باز می‌کند (یعنی غیرمتنی) ورودی یا خروجی.

شکل ۵-۱۷ | مدهای باز کردن فایل.

شی **ofstream** می‌تواند بدون باز کردن یک فایل خاص، فایلی که می‌تواند بعداً به شی الصاق شود، ایجاد گردد. برای مثال، عبارت

```
ofstream outClienFile;
```

یک شی **ofstream** بنام **outClientFile** ایجاد می‌کند. تابع عضو **open** از **ofstream** یک فایل باز کرده و آنرا به یک شی **ofstream** موجود الصاق می‌کند، همانند:

```
outClientFile.open( "clients.dat", ios::out );
```

پس از ایجاد یک شی **ofstream** و اقدام به باز کردن آن، برنامه تست می‌کند که آیا عملیات باز کردن با موفقیت همراه بوده است یا خیر. عبارت **if** در خطوط 26-22 از عملگر تابع عضو سربارگذاری شده **operator!** استفاده کرده تا تعیین کند که آیا عملیات باز کردن با موفقیت همراه شده است یا خیر. اگر **failbit** یا **badbit** برای استریم در عملیات باز کردن تنظیم شده باشد، شرط **true** برگشت خواهد داد. برخی از خطاهای که ممکن است به هنگام باز کردن فایل رخ دادند عبارتند از عدم وجود فایل برای خواندن، اقدام به باز کردن فایل برای خواندن یا نوشتن بدون مجوز و باز کردن فایلی برای نوشتن زمانیکه بر روی دیسک فضای کافی در اختیار نیست.

اگر شرط دلالت بر عدم موفقیت در باز کردن فایل داشته باشد، خط 24 پیغام خطای "File could not be opened" را چاپ کرده و خط 25 برای خاتمه دادن به برنامه تابع **exit** را فراخوانی می‌کند. آرگومانی از **exit** به محیط برگشت داده می‌شود. آرگومان صفر دلالت بر خاتمه عادی برنامه دارد، هر مقدار دیگری دلالت بر این می‌کند که برنامه به علت خطا خاتمه یافته است. محیط فراخوانی (غالبا سیستم عامل) از مقدار برگشتی توسط **exit** برای واکنش مناسب در برابر خطا استفاد می‌کنند.

یکی دیگر از عملگرهای تابع عضو سربارگذاری شده **operator void *** است که استریم را تبدیل به یک اشاره‌گر می‌کند، از اینرو می‌تواند برای تست صفر (یعنی اشاره‌گر **null**) یا غیر صفر (یعنی هر مقدار دیگر اشاره‌گر) بکار گرفته شود. زمانیکه مقدار یک اشاره‌گر بعنوان یک شرط بکار گرفته می‌شود، ++C اشاره‌گر **null** را به مقدار بولی **false** و اشاره‌گر غیر **null** را به مقدار بولی **true** تبدیل می‌کند. اگر **failbit** یا **badbit** برای استریمی تنظیم شده باشد، صفر (**false**) برگشت داده خواهد شد. شرط موجود در عبارت **while** خطوط 36-40 تابع عضو **operator void *** را بر روی تابع **cin** بصورت ضمنی اعمال می‌کند. با رسیدن به انتهای فایل، شاخص مبادرت به تنظیم **failbit** برای **cin** می‌کند. تابع **operator void *** می‌تواند برای تست شی ورودی برای انتهای فایل بجای فراخوانی صریح تابع عضو **eof** بر روی شی ورودی بکار گرفته شود.

اگر خط 19 فایل را با موفقیت باز کند، برنامه شروع به پردازش داده می‌کند. خطوط 29-28 به کاربر اعلان می‌کنند تا فیلدهای مختلف را برای هر رکورد وارد کرده یا انتهای فایل را پس از وارد کردن داده‌ها مشخص سازد. جدول شکل ۶-۱۷ حاوی ترکیبات کلیدی برای وارد کردن انتهای فایل در سیستم‌های مختلف کامپیوتری است.

خط 36 هر مجموعه از داده‌ها را استخراج کرده و تعیین می‌کند که آیا انتهای فایل وارد شده است یا خیر. زمانیکه با انتهای فایل مواجه شود یا داده نامعتبری وارد شده باشد، **operator void *** اشاره‌گر **null** را برگشت می‌دهد (که به مقدار بولی **false** تبدیل می‌کند) و عبارت **while** خاتمه می‌یابد. کاربر با وارد کردن انتهای فایل به برنامه اطلاع می‌دهد که اطلاعات دیگری برای پردازش وجود ندارد. زمانیکه کاربر کلید انتهای فایل را وارد کرد، شاخص انتهای فایل تنظیم می‌شود. حلقه عبارت **while** تا تنظیم شاخص انتهای فایل ادامه می‌یابد.

سیستم کامپیوتری	ترکیب کلیدهای صفحه کلید
UNIX/Linux/Mac OS X	*<ctrl-d>*
Microsoft Windows	*<ctrl-z>*
VAX (VMS)	*<ctrl-z>*

شکل ۱۷-٦ | کلیدهای ترکیبی نشاندهنده انتهای فایل.

خط 38 مجموعه‌ای از داده‌ها را به فایل **clients.dat** با استفاده از عملگر << و شی **outClientFile** مرتبط با فایل در ابتدای برنامه، می‌نویسد. داده‌ها را می‌توان از فایل خواند (بخش ۵-۱۷). توجه کنید بدلیل اینکه فایل ایجاد شده در شکل ۴-۱۷ یک فایل متنی ساده است، می‌توان آنرا توسط هر برنامه ویرایشگر متنی مورد بازبینی قرار داد.

زمانیکه کاربر شاخص انتهای فایل را وارد می‌کند، **main** خاتمه می‌یابد. با اینکار نابود کننده شی **outClientFile** بصورت ضمنی فراخوانی می‌شود، که مبادرت به بستن فایل **client.dat** می‌کند. همچنین برنامه می‌تواند شی **ofstream** را با استفاده از تابع عضو **close** ببندد، همانند عبارت زیر،

```
outClientFile.close();
```

در اجرای نمونه برنامه شکل ۴-۱۷ کاربر اطلاعاتی برای پنج حساب وارد کرده و با فشردن کلیدهای *ctrl*-z نشان داده که ورود اطلاعات به پایان رسیده است. این پنجره نحوه ظاهر شدن رکوردهای داده در فایل را نشان نداده است. برای بازبینی اینکه برنامه فایل را با موفقیت ایجاد کرده است، بخش بعدی نحوه خواندن این فایل و چاپ محتویات آنرا نشان داده است.

۵-۱۷ خواندن داده از یک فایل ترتیبی

فایل‌ها ذخیره کننده داده‌ها هستند، از اینرو در زمان پردازش داده‌ها نیاز است تا این داده‌ها بازیابی شوند. در بخش قبلی با نحوه ایجاد یک فایل با دسترسی ترتیبی آشنا شدید. در این بخش، با نحوه خواندن ترتیبی داده‌ها از فایل آشنا می‌شوید.

برنامه شکل ۷-۱۷ رکوردها را از فایل **clients.dat** می‌خواند، که آنرا با استفاده ازبرنامه ۴-۱۷ ایجاد کرده‌ایم و محتویات رکوردهای آنرا به نمایش در می‌آورد. با ایجاد یک شی **ifstream** یک فایل برای ورودی باز می‌شود. سازنده **ifstream** می‌تواند نام فایل و مد باز شدن فایل را به عنوان آرگومان دریافت کند. خط 31 یک شی **ifstream** بنام **inClientFile** ایجاد کرده و آنرا با فایل **clients.dat** مرتبط می‌کند. آرگومان‌های موجود در درون پرانتزها به تابع سازنده **ifstream** ارسال می‌شوند که فایل را باز کرده و یک خط ارتباطی با فایل را بنا می‌کنند.

```
1  // Fig. 17.7: Fig17_07.cpp
2  // Reading and printing a sequential file.
3  #include <iostream>
4  using std::cerr;
5  using std::cout;
6  using std::endl;
7  using std::fixed;
8  using std::ios;
9  using std::left;
10 using std::right;
11 using std::showpoint;
12
13 #include <fstream> // file stream
14 using std::ifstream; // input file stream
15
16 #include <iomanip>
17 using std::setw;
18 using std::setprecision;
19
20 #include <string>
21 using std::string;
22
23 #include <cstdlib>
24 using std::exit; // exit function prototype
25
26 void outputLine( int, const string, double ); // prototype
27
```

```
28 int main()
29 {
30    // ifstream constructor opens the file
31    ifstream inClientFile( "clients.dat", ios::in );
32
33    // exit program if ifstream could not open file
34    if ( !inClientFile )
35    {
36       cerr << "File could not be opened" << endl;
37       exit( 1 );
38    } // end if
39
40    int account;
41    char name[ 30 ];
42    double balance;
43
44    cout << left << setw( 10 ) << "Account" << setw( 13 )
45       << "Name" << "Balance" << endl << fixed << showpoint;
46
47    // display each record in file
48    while ( inClientFile >> account >> name >> balance )
49       outputLine( account, name, balance );
50
51    return 0; // ifstream destructor closes the file
52 } // end main
53
54 // display single record from file
55 void outputLine( int account, const string name, double balance )
56 {
57    cout << left << setw( 10 ) << account << setw( 13 ) << name
58       << setw( 7 ) << setprecision( 2 ) << right << balance << endl;
59 } // end function outputLine
```

```
Account    Name          Balance
100        Jones           24.98
200        Doe            345.67
300        White            0.00
400        Stone          -42.16
500        Rich           224.62
```

شکل ۷-۱۷ | خواندن و چاپ از یک فایل ترتیبی.

شی‌های از کلاس **ifstream** بطور پیش فرض برای ورودی باز می‌شوند. می‌توانیم از عبارت زیر استفاده کنیم

```
ifstream inClientFile( "clients.dat" );
```

تا **clients.dat** را برای ورودی باز کند. همانند یک شی **ofstream**، یک شی **ifstream** می‌تواند بدون باز کردن یک فایل خاص ایجاد شود، چرا که فایل می‌تواند بعدها الصاق گردد.

برنامه از شرط **!inclientFile** برای تعیین اینکه آیا فایل قبل از مبادرت به بازیابی داده‌ها از آن با موفقیت باز شده است یا خیر، استفاده کرده است. خط 48 یک مجموعه از داده‌ها (منظور رکورد است) را از فایل می‌خواند. پس از اینکه خط قبلی یک بار اجرا شد، **account** دارای مقدار 100، **name** دارای مقدار "Jones" و **balance** دارای مقدار 24.98 خواهد بود. هر بار که خط 48 اجرا می‌شود، یک رکورد دیگر از فایل را به درون متغیرهای **account**، **name** و **balance** می‌خواند. خط 49 با استفاده از تابع **outputLine** در خطوط 59-55 رکوردها را به نمایش در می‌آورد، که از دستکاری کننده‌های پارامتری شده استریم برای قالب‌بندی نمایش داده‌ها استفاده کرده است. زمانیکه به انتهای فایل می‌رسد، بطور ضمنی

عملگر * **void** در شرط **while**، اشاره‌گر **null** برگشت می‌دهد (که به مقدار بولی **false** تبدیل می‌شود)، تابع نابود کننده **ifstream** فایل را بسته و برنامه خاتمه می‌پذیرد.

برای بازیابی ترتیبی داده‌ها از یک فایل، معمولاً برنامه‌ها کار خواندن داده‌ها را از ابتدای فایل شروع کرده و بطور پیوسته داده‌ها را می‌خوانند تا اینکه به داده مورد نظر دست یابند. امکان انجام چندین باره اینکار در فایل ترتیبی (از ابتدای فایل) وجود دارد. هر دو شی **istream** و **ostream** توابع عضوی برای موقعیت‌دهی *اشاره‌گر موقعیت فایل* در نظر گرفته‌اند. این توابع عضو عبارتند از: **seekg** از **istream** و تابع **seekp** از **ostream**. هر شی **istream** دارای یک "*get pointer*" است که دلالت بر تعداد بایت‌ها در فایل از مکانی می‌کند که ورودی بعدی صورت می‌گیرد و هر شی **ostream** دارای یک "*put pointer*" است که دلالت بر تعداد بایت‌ها در فایل از مکانی می‌کند که باید خروجی بعدی جای داده شود. عبارت

```
inClientFile.seekg(0);
```

اشاره‌گر موقعیت فایل را به ابتدای فایل (موقعیت صفر) متصل به **inClientFile** انتقال می‌دهد. معمولاً آرگومان **seekg** یک مقدار صحیح **long** است. آرگومان دوم می‌تواند برای نشان دادن جهت جستجو بکار گرفته شود. جهت جستجو می‌تواند **ios::beg** (حالت پیش‌فرض) برای موقعیت‌یابی نسبی از ابتدای یک استریم، **ios::cur** برای موقعیت‌یابی نسبی با موقعیت جاری در استریم یا **ios::end** برای موقعیت‌یابی نسبی با انتهای استریم باشد. اشاره‌گر موقعیت فایل یک مقدار صحیح است که تصریح کننده مکانی در فایل بصورت عددی از بایت‌ها از مکان شروع فایل است (به اینحالت *افست* (*offset*) از ابتدای فایل گفته می‌شود). برخی از مثال‌های موقعیت‌یابی مکان اشاره‌گر فایل در زیر آورده شده است:

```
//position to the nth byte of fileObject (assumes ios::beg)
fileObject.seekg( n );

//position n bytes forward in fileObject
fileObject.seekg(n, ios::cur);

//position n bytes back from end of fileObject
fileObject.seekg( n, ios::end);

//position at end of fileObject
fileobject.seekg(0, ios::end);
```

همین عملیات‌ها را می‌توان با استفاده تابع عضو **seekp** از **ostream** انجام داد. توابع عضو **tellg** و **tellp** برای باز گرداندن مکان جاری اشاره‌گرهای "get" و "put" تدارک دیده شده‌اند. عبارت زیر مقدار اشاره‌گر موقعیت فایل "get" را به متغیر **location** از نوع **long** تخصیص می‌دهد:

```
location = fileObject.tellg();
```

برنامه شکل ۷-۱۸ به یک برنامه مدیریت اعتبار امکان می‌دهد تا اطلاعات حساب را برای مشتریانی با مانده حساب صفر (یعنی مشتریانی که به شرکت بدهکار نیستند)، مانده حساب اعتباری (منفی) و مانده حساب بدهکار (مثبت) به نمایش در آورد. برنامه یک منو به نمایش در آورده و به مدیر اعتبارات اجازه می‌دهد

تا یکی از سه گزینه را برای بدست آوردن اطلاعات اعتباری وارد سازد. گزینه1 لیستی از حساب‌ها با مانده حساب صفر، گزینه2 لیستی از حساب‌ها با مانده حساب اعتباری، گزینه3 لیستی از حساب‌های مانده بدهکار تولید می‌کند. گزینه4 به اجرای برنامه خاتمه می‌دهد. با وارد کردن یک گزینه غیرمعتبر، برنامه به کاربر اعلان می‌کند تا گزینه معتبری را وارد سازد.

```
1   // Fig. 17.8: Fig17_08.cpp
2   // Credit inquiry program.
3   #include <iostream>
4   using std::cerr;
5   using std::cin;
6   using std::cout;
7   using std::endl;
8   using std::fixed;
9   using std::ios;
10  using std::left;
11  using std::right;
12  using std::showpoint;
13
14  #include <fstream>
15  using std::ifstream;
16
17  #include <iomanip>
18  using std::setw;
19  using std::setprecision;
20
21  #include <string>
22  using std::string;
23
24  #include <cstdlib>
25  using std::exit; // exit function prototype
26
27  enum RequestType {ZERO_BALANCE=1,CREDIT_BALANCE,DEBIT_BALANCE,END};
28  int getRequest();
29  bool shouldDisplay( int, double );
30  void outputLine( int, const string, double );
31
32  int main()
33  {
34     // ifstream constructor opens the file
35     ifstream inClientFile( "clients.dat", ios::in );
36
37     // exit program if ifstream could not open file
38     if ( !inClientFile )
39     {
40        cerr << "File could not be opened" << endl;
41        exit( 1 );
42     } // end if
43
44     int request;
45     int account;
46     char name[ 30 ];
47     double balance;
48
49     // get user's request (e.g., zero, credit or debit balance)
50     request = getRequest();
51
52     // process user's request
53     while ( request != END )
54     {
55        switch ( request )
56        {
57           case ZERO_BALANCE:
58              cout << "\nAccounts with zero balances:\n";
59              break;
60           case CREDIT_BALANCE:
```

```
61             cout << "\nAccounts with credit balances:\n";
62             break;
63          case DEBIT_BALANCE:
64             cout << "\nAccounts with debit balances:\n";
65             break;
66       } // end switch
67
68       // read account, name and balance from file
69       inClientFile >> account >> name >> balance;
70
71       // display file contents (until eof)
72       while ( !inClientFile.eof() )
73       {
74          // display record
75          if ( shouldDisplay( request, balance ) )
76             outputLine( account, name, balance );
77
78          // read account, name and balance from file
79          inClientFile >> account >> name >> balance;
80       } // end inner while
81
82       inClientFile.clear();    // reset eof for next input
83       inClientFile.seekg( 0 ); // reposition to beginning of file
84       request = getRequest();  // get additional request from user
85    } // end outer while
86
87    cout << "End of run." << endl;
88    return 0; // ifstream destructor closes the file
89 } // end main
90
91 // obtain request from user
92 int getRequest()
93 {
94    int request; // request from user
95
96    // display request options
97    cout << "\nEnter request" << endl
98       << " 1 - List accounts with zero balances" << endl
99       << " 2 - List accounts with credit balances" << endl
100      << " 3 - List accounts with debit balances" << endl
101      << " 4 - End of run" << fixed << showpoint;
102
103   do // input user request
104   {
105      cout << "\n? ";
106      cin >> request;
107   } while ( request < ZERO_BALANCE && request > END );
108
109   return request;
110 } // end function getRequest
111
112 // determine whether to display given record
113 bool shouldDisplay( int type, double balance )
114 {
115    // determine whether to display zero balances
116    if ( type == ZERO_BALANCE && balance == 0 )
117       return true;
118
119    // determine whether to display credit balances
120    if ( type == CREDIT_BALANCE && balance < 0 )
121       return true;
122
123    // determine whether to display debit balances
124    if ( type == DEBIT_BALANCE && balance > 0 )
125       return true;
126
127    return false;
128 } // end function shouldDisplay
129
130 // display single record from file
```

```
131 void outputLine( int account, const string name, double balance )
132 {
133    cout << left << setw( 10 ) << account << setw( 13 ) << name
134       << setw( 7 ) << setprecision( 2 ) << right << balance << endl;
135 } // end function outputLine
```

```
Enter request
    1-  List accounts with zero balances
    2-  List accounts with credit balances
    3-  List accounts with debit balances
    4-  End of run
    ? 1

    Accounts with zero balances:
    300      White            0.00

    Enter request
    1-  List accounts with zero balances
    2-  List accounts with credit balances
    3-  List accounts with debit balances
    4-  End of run
    ? 2

    Accounts with credit balances:
    400      Stone          -42.16

    Enter request
    1-  List accounts with zero balances
    2-  List accounts with credit balances
    3-  List accounts with debit balances
    4-  End of run
    ? 3

    Accounts with debit balances:
    100      Jones           24.98
    200      Doe            345.67
    500      Rich           224.62

    Enter request
    1-  List accounts with zero balances
    2-  List accounts with credit balances
    3-  List accounts with debit balances
    4-  End of run
    ? 4
    End of run.
```

شکل ۸-۱۷ | برنامه پرس و جوی اعتبار.

۶-۱۷ به روز کردن فایل‌های ترتیبی

داده‌ای که قالب‌بندی شده و در یک فایل ترتیبی همانند برنامه شکل ۴-۱۷ نوشته شده باشد، بدون ریسک از بین رفتن سایر داده‌ها در فایل امکان تغییر و اصلاح در آن وجود ندارد. برای مثال، اگر نام "White" نیاز به تغییر به "Worthington" داشته باشد، نام قدیمی نمی‌تواند بدون معیوب ساختن فایل بازنویسی شود. رکورد White در فایل بصورت زیر نوشته شده است

```
300 White 0.00
```

اگر این رکورد از ابتدای همان مکان در فایل با استفاده از یک نام طولانی‌تر مجدداً نوشته شود، رکورد بصورت زیر در خواهد آمد

```
300 Worthingont 0.00
```

رکورد جدید حاوی شش کاراکتر بیش از رکورد قبلی است. از اینرو کاراکترهای قرار گرفته پس از دومین "o" در "Worthington" بر روی ابتدای رکورد بعدی در فایل نوشته خواهند شد. مشکل اینجاست

که، در مدل ورودی/خروجی قالب‌بندی شده از عملگرهای >> و <<، فیلدها یا رکوردها می‌توانند هر سایزی داشته باشند. برای مثال مقادیر 7, 14, 117-، 2047 و 27383 همگی از نوع صحیح هستند که در تعداد بایت یکسانی «داده خام» ذخیره می‌شوند. (در کامپیوترهای 32 بیتی، این مقدار چهار بایت است). با این وجود، این مقادیر صحیح به هنگام خروجی بعنوان متن قالب‌بندی شده به فیلدهای با سایز متفاوت تبدیل می‌شوند. بنابر این، معمولاً مدل ورودی/خروجی قالب‌بندی شده برای به روز کردن رکوردها بکار گرفته نمی‌شود.

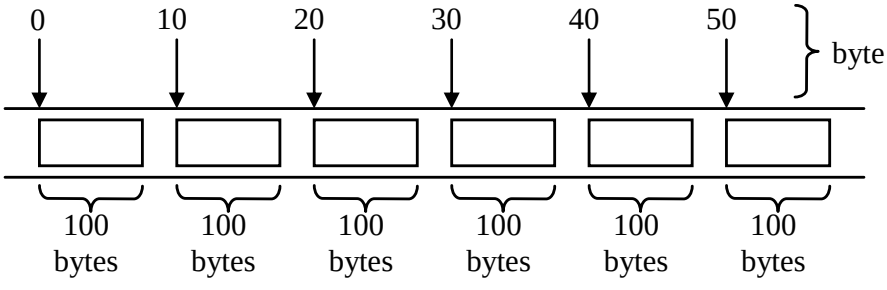
به روز کردن چنین فایلهای به روش ضعیفی صورت می‌گیرد. برای مثال، برای تغییر نام مطرح شده در فوق، می‌توان رکوردهای قبل از 300 White 0.00 در یک فایل ترتیبی را به یک فایل جدید کپی کرده، سپس رکورد به روز شده را به فایل جدید بنویسیم و رکوردهای پس از 300 White 0.00 را به فایل جدید کپی نمائیم. این فرآیند برای به روز کردن هر رکوردی در فایل لازم است.

۱۷-۷ فایل با دسترسی تصادفی

تا بدین جا به توضیح نحوه ایجاد فایل‌ها با دسترسی ترتیبی و جستجو در آنها پرداخته‌ایم. با این همه، فایل‌ها با دسترسی ترتیبی برای برنامه‌های که نیاز به "دسترسی آنی" دارند مناسب نیستند. در چنین برنامه‌هایی باید سرعت و بطور آنی به اطلاعات یک رکورد خاص دسترسی پیدا کرد. از جمله برنامه‌های دسترسی آنی می‌توان سیستم‌های رزرو خطوط هوایی، سیستم‌های بانکی، سیستم‌های فروش، ماشین‌های پرداخت اتوماتیک و دیگر سیستم‌های پردازش تراکنشی نام برد، که مستلزم دسترسی سریع به داده خاصی هستند. در یک سیستم بانکی ممکن است حساب صدها، هزاران یا میلیون‌ها مشتری وجود داشته باشد، در چنین سیستمی باید در عرض چند ثانیه حساب مورد نظر پیدا شود. چنین دسترسی آنی به کمک فایل‌های تصادفی امکان‌پذیر است. به رکوردهای متمایز در یک فایل با دسترسی تصادفی می‌توان بطور مستقیم و بسرعت دسترسی پیدا کرد بدون اینکه جستجوهای زیادی در میان رکوردهای دیگر انجام داد که در فایل‌هایی با دسترسی ترتیبی انجام آن ضروری است. فایل‌ها با دسترسی تصادفی با عنوان فایل‌هایی با دسترسی مستقیم نیز شناخته می‌شوند.

همانطوری که در اوایل این فصل هم گفته شد، C++ ساختار خاصی بر روی فایل‌ها اعمال نمی‌کند، از اینرو برنامه‌هایی که از فایل‌های تصادفی استفاده می‌کنند، بایستی از قابلیت دسترسی تصادفی برخوردار باشند. تکنیک‌های گوناگونی برای ایجاد فایل‌های تصادفی وجود دارد. شاید ساده‌ترین روش این باشد که تمام رکوردها دارای طول ثابت در فایل باشند. به هنگام استفاده از رکوردهایی با طول ثابت، برنامه می‌تواند مکان دقیق هر رکورد را با توجه به ابتدای فایل محاسبه کند.

در شکل ۹-۱۷ می‌توانید یک فایل با دسترسی تصادفی که دارای رکوردهایی با طول ثابت است، مشاهده کنید (هر رکورد در این تصویر ۱۰۰ بایت است). داده می‌تواند بدون اینکه داده دیگری را در فایل از بین ببرد، وارد یک فایل تصادفی شود. علاوه بر این، می‌توان داده ذخیره شده را به روز یا حذف کرد، بدون اینکه کل فایل مجدداً نوشته شود. در بخش‌های بعدی، با نحوه ایجاد فایل تصادفی، نوشتن داده به فایل، خواندن داده بصورت تصادفی و ترتیبی، به روز کردن و حذف داده‌های که به آنها نیاز نیست، آشنا خواهید شد.



شکل ۹-۱۷ | فایل با دسترسی تصادفی با رکوردهای طول ثابت.

۸-۱۷ ایجاد فایل تصادفی

تابع عضو **wtire** از **ostream** به تعداد ثابتی بایت، از ابتدای یک مکان مشخص شده در حافظه به استریم تصریح شده، در خروجی قرار می‌دهد. زمانیکه استریم با فایلی مرتبط شد، تابع **write** داده را در مکانی از فایل که توسط اشاره‌گر موقعیت فایل مشخص شده است، می‌نویسد. تابع **read** از **istream** به تعداد ثابتی بایت از استریم مشخص را به یک ناحیه در حافظه از ابتدای آدرس تعیین شده وارد می‌کند. اگر استریم با فایلی مرتبط شده باشد، تابع **read** بایت‌ها را در مکانی از فایل که توسط اشاره‌گر موقعیت فایل مشخص شده است، وارد می‌کند.

نوشتن بایت‌ها با تابع عضو write از ostream

به هنگام نوشتن **number** از نوع صحیح به یک فایل، بجای استفاده از عبارت

```
outFile << number;
```

که برای یک مقدار صحیح چهار بایتی می‌تواند تعدادی رقم بصورت یک یا چندین 11 چاپ کند، می‌توانیم از عبارت زیر استفاده کنیم

```
outFile.write( reinterpret_cast< const char * >( &number ),
   sizeof( number ) );
```

که همیشه نسخه باینری از مقدار صحیح چهار بایتی را می‌نویسد. تابع **write** با اولین آرگومان خود بصورت گروهی از بایت‌ها بواسطه شی در حافظه یک **const char *** که یک اشاره‌گر به یک بایت است (بخاطر داشته باشید که **char** یک بایتی است) رفتار می‌کند. با شروع از موقعیت مشخص شده، تابع **write**

تعداد بایت تعیین شده توسط آرگومان دوم را خارج می‌سازد، یک مقدار صحیح از نوع **size_t**. همانطور که مشاهده خواهید کرد، تابع **read** می‌تواند متعاقبا برای خواندن چهار بایت و قرار دادن آنها در متغیر **number** بکار گرفته شود.

تبدیل مابین انواع اشاره‌گر با عملگر reinterpret_cast

متاسفانه، اغلب اشاره‌گرهای که به تابع **write** بعنوان اولین آرگومان ارسال می‌کنیم از نوع **const char *** نیستند. برای خارج ساختن شی‌ها از انواع دیگر، بایستی اشاره‌گر به این شی‌ها را به نوع **const char *** تبدیل کنیم، در غیر اینصورت کامپایلر قادر به کامپایل فراخوانی تابع **write** نخواهد بود. C++ عملگر ***reinterpret_cast*** را برای چنین مواردی آماده کرده است. همچنین می‌توانید از این عملگر تبدیل برای تبدیل مابین اشاره‌گر و نوع‌های صحیح و برعکس استفاده کنید. بدون حضور **reinterpret_cast**، عبارت **write** که می‌خواهد **number** را خارج سازد، کامپایل نخواهد شد، چرا که کامپایلر اجازه نمی‌دهد تا اشاره‌گر از نوع **int *** (نوع برگشتی توسط عبارت **&number**) به تابعی ارسال شود که در انتظار آرگومانی از نوع **const char *** است، تا آنجا که به کامپایلر مربوط می‌شود، این نوع‌ها با هم سازگار نیستند.

عملگر **reinterpret_cast** در زمان کامپایل وارد عمل می‌شود و تغییری در مقدار شی که عملوند آن به آن اشاره می‌کند بوجود نمی‌آورد. بجای آن، از کامپایلر تقاضا می‌کند تا عملوند را بعنوان نوع هدف دوباره تفسیر کند (مشخص شده در درون < > که پس از کلمه کلیدی **reinterpret_cast** آورده می‌شود). در برنامه شکل ۱۲-۱۷ از این عملگر برای تبدیل یک اشاره‌گر **ClientData** به یک **const char *** استفاده کرده‌ایم، که شی **ClientData** را بصورت بایت‌های در خروجی یک فایل مجدداً تفسیر می‌کند. برنامه پردازش فایل با دسترسی تصادفی بندرت یک فیلد منفرد را در یک فایل می‌نویسد. معمولاً، آنها یک شی از یک کلاس را در هر بار می‌نویسند که در مثال‌های بعدی شاهد آن خواهید بود.

برنامه پردازش اعتبار

به صورت مسئله زیر توجه کنید:

یک برنامه پردازش کننده اعتبار بنویسید که قادر به ذخیره‌سازی بیش از صد رکورد با طول ثابت برای شرکتی باشد که می‌تواند بیش از صد مشتری داشته باشد. هر رکورد باید متشکل از یک شماره حساب (که همانند کلید رکورد عمل کند)، نام خانوادگی، نام و موجودی باشد. برنامه بایستی قادر به، به روز کردن حساب، وارد کردن حساب‌های جدید، حذف حساب و وارد ساختن کل رکوردهای حساب به یک فایل متنی قالب‌بندی شده با هدف چاپ باشد.

در چند بخش بعدی به معرفی تکنیک‌های بکار رفته در این برنامه خواهیم پرداخت. برنامه شکل ۱۲-۱۷ به بیان نحوه باز کردن یک فایل تصادفی، تعریف فرمت رکورد با استفاده از شی از کلاس **ClientData** (شکل‌های ۱۰-۱۷ و ۱۱-۱۷) و نوشتن داده به دیسک با فرمت باینری، پرداخته است.

این برنامه مبادرت به مقداردهی اولیه تمام صد رکورد از فایل **credit.dat** با شی‌های تهی و با استفاده از تابع **write** می‌کند هر شی تهی حاوی صفر برای شماره حساب، رشته **null** (که توسط جفت گوتیشن خالی مشخص می‌شود) برای نام خانوادگی و نام، 0.0 برای موجودی است.

```
// Fig. 17.10: ClientData.h
// Class ClientData definition used in Fig. 17.12-Fig. 17.15.
#ifndef CLIENTDATA_H
#define CLIENTDATA_H

#include <string>
using std::string;

class ClientData
{
public:
   // default ClientData constructor
   ClientData( int = 0, string = "", string = "", double = 0.0 );

   // accessor functions for accountNumber
   void setAccountNumber( int );
   int getAccountNumber() const;

   // accessor functions for lastName
   void setLastName( string );
   string getLastName() const;

   // accessor functions for firstName
   void setFirstName( string );
   string getFirstName() const;

   // accessor functions for balance
   void setBalance( double );
   double getBalance() const;
private:
   int accountNumber;
   char lastName[ 15 ];
   char firstName[ 10 ];
   double balance;
}; // end class ClientData

#endif
```

شکل ۱۰-۱۷ | فایل سرآیند ClientData.

شی‌ها از کلاس **string** دارای سایز واحدی نیستند چرا که از روش اخذ حافظه دینامیکی برای جا دادن رشته‌ها با طول‌های متفاوت استفاده می‌کنند. بایستی این برنامه رکوردهای با طول ثابت داشته باشد، از اینرو کلاس **ClientData** نام و نام خانوادگی مشتری را در آرایه‌های **char** با طول ثابت ذخیره می‌کند. توابع عضو **setLastName** (شکل ۱۱-۱۷، خطوط 45-37) و **setFirstName** (خطوط 62-54 از شکل ۱۱-۱۷) هر یک مبادرت به کپی کاراکترها از یک شی رشته بدون آرایه **char** متناظر می‌کنند. به تابع **setLanstName** توجه کنید. خط 40 مبادرت به مقداردهی اولیه **const char * lastNameValue** با نتیجه فراخوانی تابع عضو **data** می‌کند که یک آرایه حاوی کاراکترهای از رشته برگشت می‌دهد. خط 41 تابع عضو **size** را برای بدست آوردن طول رشته **lastNameString** فراخوانی می‌کند. خط 42 مطمئن می‌شود که **length** (طول) کمتر از 25 کاراکتر است، سپس خط 43 طول کاراکترها را از

lastNameValue به آرایه **lastName** کپی می‌کند. تابع عضو **setFirstName** همین مراحل را برای نام انجام می‌دهد.

```
1  // Fig. 17.11: ClientData.cpp
2  // Class ClientData stores customer's credit information.
3  #include <string>
4  using std::string;
5
6  #include "ClientData.h"
7
8  // default ClientData constructor
9  ClientData::ClientData( int accountNumberValue,
10    string lastNameValue,string firstNameValue,double balanceValue )
11 {
12    setAccountNumber( accountNumberValue );
13    setLastName( lastNameValue );
14    setFirstName( firstNameValue );
15    setBalance( balanceValue );
16 } // end ClientData constructor
17
18 // get account-number value
19 int ClientData::getAccountNumber() const
20 {
21    return accountNumber;
22 } // end function getAccountNumber
23
24 // set account-number value
25 void ClientData::setAccountNumber( int accountNumberValue )
26 {
27    accountNumber = accountNumberValue; // should validate
28 } // end function setAccountNumber
29
30 // get last-name value
31 string ClientData::getLastName() const
32 {
33    return lastName;
34 } // end function getLastName
35
36 // set last-name value
37 void ClientData::setLastName( string lastNameString )
38 {
39    // copy at most 15 characters from string to lastName
40    const char *lastNameValue = lastNameString.data();
41    int length = lastNameString.size();
42    length = ( length < 15 ? length : 14 );
43    strncpy( lastName, lastNameValue, length );
44    lastName[ length ] = '\0'; // append null character to lastName
45 } // end function setLastName
46
47 // get first-name value
48 string ClientData::getFirstName() const
49 {
50    return firstName;
51 } // end function getFirstName
52
53 // set first-name value
54 void ClientData::setFirstName( string firstNameString )
55 {
56    // copy at most 10 characters from string to firstName
57    const char *firstNameValue = firstNameString.data();
58    int length = firstNameString.size();
59    length = ( length < 10 ? length : 9 );
60    strncpy( firstName, firstNameValue, length );
61    firstName[ length ] = '\0'; // append null character to firstName
62 } // end function setFirstName
63
64 // get balance value
65 double ClientData::getBalance() const
```

```
66 {
67    return balance;
68 } // end function getBalance
69
70 // set balance value
71 void ClientData::setBalance( double balanceValue )
72 {
73    balance = balanceValue;
74 } // end function setBalance
```

شکل ۱۱-۱۷ | کلاس ClientData عرضه کننده اطلاعات اعتباری مشتری.

در شکل ۱۲-۱۷، خط 18 یک شی از **ofstream** برای فایل **credit.dat** ایجاد می‌کند. آرگومان دوم در سازنده، **ios::binary**، بر این نکته دلالت دارد که فایل را برای خروجی در مد باینری باز کرده‌ایم، که برای نوشتن رکوردهای با طول ثابت در فایل ضروری است. خطوط 32-31 سبب می‌شوند که **blankClient** در فایل **credit.dat** مرتبط با شی **outCredit** نوشته شود. بخاطر داشته باشید که عملگر **sizeof** سایز شی احاطه شده در درون پرانتز را برحسب بایت برگشت می‌دهد. آرگومان اول در تابع **write**، خط 31 بایستی از نوع **const char *** باشد. اما نوع داده **&blankClient** از نوع **ClientData *** است. برای تبدیل **&blankClient** به **const char ***، خط 31 از عملگر تبدیل **reinterpret_cast** استفاده کرده است. از اینرو فراخوانی **write** بدون اینکه خطای کامپایل بدنبال داشته باشد، صورت می‌گیرد.

```
1  // Fig. 17.12: Fig17_12.cpp
2  // Creating a randomly accessed file.
3  #include <iostream>
4  using std::cerr;
5  using std::endl;
6  using std::ios;
7
8  #include <fstream>
9  using std::ofstream;
10
11 #include <cstdlib>
12 using std::exit; // exit function prototype
13
14 #include "ClientData.h" // ClientData class definition
15
16 int main()
17 {
18    ofstream outCredit( "credit.dat", ios::binary );
19
20    // exit program if ofstream could not open file
21    if ( !outCredit )
22    {
23       cerr << "File could not be opened." << endl;
24       exit( 1 );
25    } // end if
26
27    ClientData blankClient; // constructor zeros out each data member
28
29    // output 100 blank records to file
30    for ( int i = 0; i < 100; i++ )
31     outCredit.write(reinterpret_cast< const char *>(&blankClient),
32          sizeof( ClientData ) );
33
34    return 0;
35 } // end main
```

شکل ۱۲-۱۷ | ایجاد فایل با دسترسی تصادفی با 100 رکورد خالی پشت سرهم.

۹-۱۷ نوشتن داده بصورت تصادفی در فایل با دسترسی تصادفی

برنامه شکل ۱۳-۱۷ مبادرت به نوشتن داده به فایل **credit.dat** کرده و از توابع **seekp** و **write** برای ذخیره‌سازی داده در مکان مشخص شده در فایل استفاده می‌کند. تابع **seekp** اشاره‌گر موقعیت فایل را در مکان مشخص در فایل قرار می‌دهد، سپس تابع **write** داده را می‌نویسد. دقت کنید که خط 19 شامل فایل سرآیند *ClientData.h* تعریف شده در شکل ۱۰-۱۷ است، از اینرو برنامه می‌تواند از شی‌های **ClientData** استفاده کند.

```
1  // Fig. 17.13: Fig17_13.cpp
2  // Writing to a random-access file.
3  #include <iostream>
4  using std::cerr;
5  using std::cin;
6  using std::cout;
7  using std::endl;
8  using std::ios;
9
10 #include <iomanip>
11 using std::setw;
12
13 #include <fstream>
14 using std::fstream;
15
16 #include <cstdlib>
17 using std::exit; // exit function prototype
18
19 #include "ClientData.h" // ClientData class definition
20
21 int main()
22 {
23    int accountNumber;
24    char lastName[ 15 ];
25    char firstName[ 10 ];
26    double balance;
27
28    fstream outCredit("credit.dat",ios::in | ios::out | ios::binary );
29
30    // exit program if fstream cannot open file
31    if ( !outCredit )
32    {
33       cerr << "File could not be opened." << endl;
34       exit( 1 );
35    } // end if
36
37    cout << "Enter account number (1 to 100, 0 to end input)\n? ";
38
39    // require user to specify account number
40    ClientData client;
41    cin >> accountNumber;
42
43    // user enters information, which is copied into file
44    while ( accountNumber > 0 && accountNumber <= 100 )
45    {
46       // user enters last name, first name and balance
47       cout << "Enter lastname, firstname, balance\n? ";
48       cin >> setw( 15 ) >> lastName;
49       cin >> setw( 10 ) >> firstName;
50       cin >> balance;
51
52       // set record accountNumber, lastName, firstName and balance values
53       client.setAccountNumber( accountNumber );
54       client.setLastName( lastName );
55       client.setFirstName( firstName );
56       client.setBalance( balance );
```

```
57
58       // seek position in file of user-specified record
59       outCredit.seekp( ( client.getAccountNumber() - 1 ) *
60          sizeof( ClientData ) );
61
62       // write user-specified information in file
63       outCredit.write( reinterpret_cast< const char * >( &client ),
64          sizeof( ClientData ) );
65
66       // enable user to enter another account
67       cout << "Enter account number\n? ";
68       cin >> accountNumber;
69    } // end while
70
71    return 0;
72 } // end main
```

```
Enter account number (1 to 100, 0 to end input)
? 37
Enter lastname, firstname, balance
? Barker Doug 0.00
Enter account number
? 29
Enter lastname, firstname, balance
? Brown Nancy -24.54
Enter account number
?96
Enter lastname, firstname, balance
? Stone Sam 34.98
Enter account number
? 88
Enter lastname, firstname, balance
? Smith Dave 258.34
Enter account number
? 33
Enter lastname, firstname, balance
? Dunn Stacey 314.33
Enter account number
? 0
```

شکل ۱۳-۱۷ | نوشتن داده در فایل با دسترسی تصادفی.

خطوط 60-59 اشاره‌گر موقعیت فایل را برای شی **outCredit** برحسب بایت و با استفاده از محاسبه زیر انتقال می‌دهند

```
( client.getAccountNumber() - 1) * sizeof( ClientData )
```

چون شماره حساب مابین 1 و 100 است، 1 به هنگام محاسبه موقعیت بایت رکورد به هنگام محاسبه کسر می‌شود. از اینرو، برای رکورد 1، اشاره‌گر موقعیت فایل با بایت صفر در فایل تنظیم می‌شود. در خط 28 از شی **outCredit** برای باز کردن فایل **credit.data** استفاده شده است. فایل در مد باینری برای خروجی و ورودی باز شده است که ترکیبی از مدهای **ios::in**، **ios::out** و **ios:binary** است. با استفاده از عملگر **OR** انحصاری (|) می‌توان چندین مد باز کردن فایل را در کنار هم بکار گرفت. باز کردن فایل موجود **credit.dat** به این روش، ما را مطمئن می‌سازد که این برنامه می‌تواند رکوردهای نوشته شده در فایل توسط برنامه ۱۲-۱۷ را نگهداری کند، بجای اینکه فایل را از اول ایجاد کنیم.

۱۰-۱۷ خواندن داده از فایل با دسترسی تصادفی بفرم ترتیبی

در بخش‌های قبلی، یک فایل با دسترسی تصادفی ایجاد کرده و داده‌های در آن فایل نوشتیم. در این بخش برنامه‌ای ایجاد می‌کنیم که فایل را بصورت ترتیبی یا پشت سرهم خوانده و فقط رکوردهای که حاوی اطلاعات هستند چاپ کند.

تابع **read** تعداد بایت‌های مشخص شده از موقعیت جاری در استریم تصریح شده را وارد می‌سازد. برای مثال، خطوط 58-57 از شکل ۱۴-۱۷ تعداد بایت‌های مشخص شده توسط **sizeof(ClientData)** را از طریق **inCredit** خوانده و داده را در رکورد **client** ذخیره می‌سازد. توجه کنید که تابع **read** مستلزم آن است که آرگومان اول از نوع *** char** باشد. از آنجا که **&Client** از نوع *** ClientData** است بایستی با استفاده از عملگر تبدیل **reinterpret_cast** تبدیل به *** char** شود. خط 24 شامل فایل سرآیند ***clientData.h*** تعریف شده در شکل ۱۰-۱۷ است و از اینرو برنامه می‌تواند از شی‌های **ClientData** استفاده کند.

برنامه شکل ۱۴-۱۷ بصورت ترتیبی هر رکورد موجود در فایل **credit.dat** را می‌خواند و بررسی می‌کند که آیا رکورد حاوی داده است یا خیر و رکوردهای حاوی داده را بصورت قالب‌بندی شده به نمایش در می‌آورد. شرط موجود در خط 50 از تابع عضو **eof** برای تعیین اینکه به انتهای فایل رسیده است یا خیر استفاده می‌کند و سبب می‌شود تا اجرای عبارت **while** خاتمه پذیرد. همچنین اگر خطای به هنگام خواندن از فایل رخ دهد، حلقه خاتمه می‌یابد، چرا که **inCredit** با **false** ارزیابی می‌شود. داده وارد شده به فایل توسط تابع **outputLine** خارج می‌شود (خطوط 72-65) که دو آرگومان دریافت می‌کند، یک شی **ostream** و یک ساختار **clientData** برای خروجی. نوع پارامتر **ostream** جالب توجه است چرا که هر شی **ostream** (همانند **out**) یا هر شی از کلاس مشتق شده از **ostream** (همانند یک شی از نوع **ofstream**) می‌تواند بعنوان آرگومان در نظر گرفته شود. به این معنی که همان تابع می‌تواند بکار گرفته شود.

```
1  // Fig. 17.14: Fig17_14.cpp
2  // Reading a random access file sequentially.
3  #include <iostream>
4  using std::cerr;
5  using std::cout;
6  using std::endl;
7  using std::fixed;
8  using std::ios;
9  using std::left;
10 using std::right;
11 using std::showpoint;
12
13 #include <iomanip>
14 using std::setprecision;
15 using std::setw;
16
17 #include <fstream>
18 using std::ifstream;
19 using std::ostream;
20
```

```
21 #include <cstdlib>
22 using std::exit; // exit function prototype
23
24 #include "ClientData.h" // ClientData class definition
25
26 void outputLine( ostream&, const ClientData & ); // prototype
27
28 int main()
29 {
30    ifstream inCredit( "credit.dat", ios::in );
31
32    // exit program if ifstream cannot open file
33    if ( !inCredit )
34    {
35       cerr << "File could not be opened." << endl;
36       exit( 1 );
37    } // end if
38
39    cout << left << setw( 10 ) << "Account" << setw( 16 )
40       << "Last Name" << setw( 11 ) << "First Name" << left
41       << setw( 10 ) << right << "Balance" << endl;
42
43    ClientData client; // create record
44
45    // read first record from file
46    inCredit.read( reinterpret_cast< char * >( &client ),
47       sizeof( ClientData ) );
48
49    // read all records from file
50    while ( inCredit && !inCredit.eof() )
51    {
52       // display record
53       if ( client.getAccountNumber() != 0 )
54          outputLine( cout, client );
55
56       // read next from file
57       inCredit.read( reinterpret_cast< char * >( &client ),
58          sizeof( ClientData ) );
59    } // end while
60
61    return 0;
62 } // end main
63
64 // display single record
65 void outputLine( ostream &output, const ClientData &record )
66 {
67    output << left << setw( 10 ) << record.getAccountNumber()
68       << setw( 16 ) << record.getLastName()
69       << setw( 11 ) << record.getFirstName()
70       << setw( 10 ) << setprecision( 2 ) << right << fixed
71       << showpoint << record.getBalance() << endl;
72 } // end function outputLine
```

```
Account    Last Name       First Name      Balance
29         Brown           Nancy            -24.54
33         Dunn            Stacey           314.33
37         Barker          Doug               0.00
88         Smith           Dave             258.34
96         Stone           Sam               34.98
```

شکل ۱۷-۱۴ | خواندن از یک فایل تصادفی بصورت ترتیبی.

۱۷-۱۱ مبحث آموزشی: برنامه پردازش تراکنشی

در این بخش به معرفی اصول یک برنامه پردازش تراکنشی (شکل ۱۷-۱۵) با استفاده از یک فایل تصادفی می‌پردازیم که پردازش‌های را با دسترسی فوری انجام می‌دهد. این برنامه اطلاعات حساب بانکی را در خود نگهداری می‌کند. برنامه قادر به، به روز کردن حساب‌های موجود، افزودن حساب‌های جدید، حذف

حساب و ذخیره‌سازی تمام حساب‌های جاری بصورت یک لیست قالب‌بندی شده در یک فایل متنی است. فرض می‌کنیم که برنامه شکل ۱۲–۱۷ برای ایجاد فایل **credit.dat** اجرا شده است و برنامه شکل ۱۳–۱۷ هم برای وارد ساختن داده‌های اولیه بکار گرفته شده باشد.

```
1  // Fig. 17.15: Fig17_15.cpp
2  // This program reads a random access file sequentially, updates
3  // data previously written to the file, creates data to be placed
4  // in the file, and deletes data previously in the file.
5  #include <iostream>
6  using std::cerr;
7  using std::cin;
8  using std::cout;
9  using std::endl;
10 using std::fixed;
11 using std::ios;
12 using std::left;
13 using std::right;
14 using std::showpoint;
15
16 #include <fstream>
17 using std::ofstream;
18 using std::ostream;
19 using std::fstream;
20
21 #include <iomanip>
22 using std::setw;
23 using std::setprecision;
24
25 #include <cstdlib>
26 using std::exit; // exit function prototype
27
28 #include "ClientData.h" // ClientData class definition
29
30 int enterChoice();
31 void createTextFile( fstream& );
32 void updateRecord( fstream& );
33 void newRecord( fstream& );
34 void deleteRecord( fstream& );
35 void outputLine( ostream&, const ClientData & );
36 int getAccount( const char * const );
37
38 enum Choices { PRINT = 1, UPDATE, NEW, DELETE, END };
39
40 int main()
41 {
42    // open file for reading and writing
43    fstream inOutCredit( "credit.dat", ios::in | ios::out );
44
45    // exit program if fstream cannot open file
46    if ( !inOutCredit )
47    {
48       cerr << "File could not be opened." << endl;
49       exit ( 1 );
50    } // end if
51
52    int choice; // store user choice
53
54    // enable user to specify action
55    while ( ( choice = enterChoice() ) != END )
56    {
57       switch ( choice )
58       {
59          case PRINT: // create text file from record file
60             createTextFile( inOutCredit );
61             break;
62          case UPDATE: // update record
63             updateRecord( inOutCredit );
```

```
64             break;
65          case NEW: // create record
66             newRecord( inOutCredit );
67             break;
68          case DELETE: // delete existing record
69             deleteRecord( inOutCredit );
70             break;
71          default://display error if user does not select valid choice
72             cerr << "Incorrect choice" << endl;
73             break;
74       } // end switch
75
76       inOutCredit.clear(); // reset end-of-file indicator
77    } // end while
78
79    return 0;
80 } // end main
81
82 // enable user to input menu choice
83 int enterChoice()
84 {
85    // display available options
86    cout << "\nEnter your choice" << endl
87       << "1 - store a formatted text file of accounts" << endl
88       << "    called \"print.txt\" for printing" << endl
89       << "2 - update an account" << endl
90       << "3 - add a new account" << endl
91       << "4 - delete an account" << endl
92       << "5 - end program\n? ";
93
94    int menuChoice;
95    cin >> menuChoice; // input menu selection from user
96    return menuChoice;
97 } // end function enterChoice
98
99 // create formatted text file for printing
100 void createTextFile( fstream &readFromFile )
101 {
102    // create text file
103    ofstream outPrintFile( "print.txt", ios::out );
104
105    // exit program if ofstream cannot create file
106    if ( !outPrintFile )
107    {
108       cerr << "File could not be created." << endl;
109       exit( 1 );
110    } // end if
111
112    outPrintFile << left << setw( 10 ) << "Account" << setw( 16 )
113       << "Last Name" << setw( 11 ) << "First Name" << right
114       << setw( 10 ) << "Balance" << endl;
115
116    // set file-position pointer to beginning of readFromFile
117    readFromFile.seekg( 0 );
118
119    // read first record from record file
120    ClientData client;
121    readFromFile.read( reinterpret_cast< char * >( &client ),
122       sizeof( ClientData ) );
123
124    // copy all records from record file into text file
125    while ( !readFromFile.eof() )
126    {
127       // write single record to text file
128       if ( client.getAccountNumber() != 0 ) // skip empty records
129          outputLine( outPrintFile, client );
130
131       // read next record from record file
132       readFromFile.read( reinterpret_cast< char * >( &client ),
133          sizeof( ClientData ) );
```

```
134    } // end while
135 } // end function createTextFile
136
137 // update balance in record
138 void updateRecord( fstream &updateFile )
139 {
140    // obtain number of account to update
141    int accountNumber = getAccount( "Enter account to update" );
142
143    // move file-position pointer to correct record in file
144    updateFile.seekg( ( accountNumber - 1 ) * sizeof( ClientData ) );
145
146    // read first record from file
147    ClientData client;
148    updateFile.read( reinterpret_cast< char * >( &client ),
149       sizeof( ClientData ) );
150
151    // update record
152    if ( client.getAccountNumber() != 0 )
153    {
154       outputLine( cout, client ); // display the record
155
156       // request user to specify transaction
157       cout << "\nEnter charge (+) or payment (-): ";
158       double transaction; // charge or payment
159       cin >> transaction;
160
161       // update record balance
162       double oldBalance = client.getBalance();
163       client.setBalance( oldBalance + transaction );
164       outputLine( cout, client ); // display the record
165
166       // move file-position pointer to correct record in file
167       updateFile.seekp((accountNumber - 1) * sizeof( ClientData ) );
168
169       // write updated record over old record in file
170       updateFile.write( reinterpret_cast< const char * >( &client ),
171          sizeof( ClientData ) );
172    } // end if
173    else // display error if account does not exist
174       cerr << "Account #" << accountNumber
175          << " has no information." << endl;
176 } // end function updateRecord
177
178 // create and insert record
179 void newRecord( fstream &insertInFile )
180 {
181    // obtain number of account to create
182    int accountNumber = getAccount( "Enter new account number" );
183
184    // move file-position pointer to correct record in file
185    insertInFile.seekg((accountNumber - 1) * sizeof( ClientData ) );
186
187    // read record from file
188    ClientData client;
189    insertInFile.read( reinterpret_cast< char * >( &client ),
190       sizeof( ClientData ) );
191
192    // create record, if record does not previously exist
193    if ( client.getAccountNumber() == 0 )
194    {
195       char lastName[ 15 ];
196       char firstName[ 10 ];
197       double balance;
198
199       // user enters last name, first name and balance
200       cout << "Enter lastname, firstname, balance\n? ";
201       cin >> setw( 15 ) >> lastName;
202       cin >> setw( 10 ) >> firstName;
203       cin >> balance;
```

```
204
205       // use values to populate account values
206       client.setLastName( lastName );
207       client.setFirstName( firstName );
208       client.setBalance( balance );
209       client.setAccountNumber( accountNumber );
210
211       // move file-position pointer to correct record in file
212       insertInFile.seekp((accountNumber - 1) * sizeof(ClientData) );
213
214       // insert record in file
215       insertInFile.write(reinterpret_cast<const char * >( &client ),
216          sizeof( ClientData ) );
217    } // end if
218    else // display error if account already exists
219       cerr << "Account #" << accountNumber
220          << " already contains information." << endl;
221 } // end function newRecord
222
223 // delete an existing record
224 void deleteRecord( fstream &deleteFromFile )
225 {
226    // obtain number of account to delete
227    int accountNumber = getAccount( "Enter account to delete" );
228
229    // move file-position pointer to correct record in file
230    deleteFromFile.seekg(( accountNumber - 1) * sizeof(ClientData));
231
232    // read record from file
233    ClientData client;
234    deleteFromFile.read( reinterpret_cast< char * >( &client ),
235       sizeof( ClientData ) );
236
237    // delete record, if record exists in file
238    if ( client.getAccountNumber() != 0 )
239    {
240       ClientData blankClient; // create blank record
241
242       // move file-position pointer to correct record in file
243       deleteFromFile.seekp( ( accountNumber - 1 ) *
244          sizeof( ClientData ) );
245
246       // replace existing record with blank record
247       deleteFromFile.write(
248          reinterpret_cast< const char * >( &blankClient ),
249          sizeof( ClientData ) );
250
251       cout << "Account #" << accountNumber << " deleted.\n";
252    } // end if
253    else // display error if record does not exist
254       cerr << "Account #" << accountNumber << " is empty.\n";
255 } // end deleteRecord
256
257 // display single record
258 void outputLine( ostream &output, const ClientData &record )
259 {
260    output << left << setw( 10 ) << record.getAccountNumber()
261       << setw( 16 ) << record.getLastName()
262       << setw( 11 ) << record.getFirstName()
263       << setw( 10 ) << setprecision( 2 ) << right << fixed
264       << showpoint << record.getBalance() << endl;
265 } // end function outputLine
266
267 // obtain account-number value from user
268 int getAccount( const char * const prompt )
269 {
270    int accountNumber;
271
272    // obtain account-number value
273    do
```

```
274    {
275       cout << prompt << " (1 - 100): ";
276       cin >> accountNumber;
277    } while ( accountNumber < 1 || accountNumber > 100 );
278
279    return accountNumber;
280 } // end function getAccount
```

شکل ۱۵-۱۷ | برنامه حساب بانکی.

برنامه دارای پنج گزینه است (گزینه5 برای خاتمه دادن به برنامه). گزینه1 تابع **createTextFile** را برای ذخیره یک لیست قالب‌بندی شده از تمام اطلاعات حساب در یک فایل متنی بنام **print.txt** که می‌توان از آن چاپ گرفت، فراخوانی می‌کند. تابع **createTextFile** در خطوط 135-100 یک شی از **fstream** بعنوان آرگومان دریافت و برای وارد کردن داده از فایل **credit.dat** بکار می‌گیرد. تابع **createTextFile** تابع عضو **read** را فراخوانی کرده (خطوط 133-132) و از تکنیک دسترسی پشت سرهم یا ترتیبی فایل (شکل ۱۴-۱۷) برای وارد کردن داده از **credit.dat** استفاده می‌کند. تابع **outputLine** که در بخش ۱۰-۱۷ توضیح داده شده است، برای نوشتن داده در فایل **print.txt** بکار گرفته شده است. توجه کنید تابع **createTextFile** از تابع عضو **seekg** در خط 117 برای اطمینان از اینکه اشاره‌گر موقعیت فایل در ابتدای فایل قرار گرفته باشد، استفاد کرده است. پس از انتخاب گزینه**1**، فایل **print.txt** حاوی داده‌های زیر خواهد بود

Account	Last Name	First Name	Balance
29	Brown	Nancy	-24.54
33	Bahram	Stacey	314.33
37	Barker	Doug	0.00
88	Smith	Dave	258.34
96	Stone	Sam	34.98

گزینه2 تابع **updateRecord** را برای به روز کردن یک حساب فراخوانی می‌کند (خطوط 176-138). این تابع فقط یک رکورد موجود را به روز می‌کند از اینرو، ابتدا تابع تعیین می‌کند که آیا رکورد مشخص شده خالی است یا خیر. خطوط 149-148 داده را بدرون شی **client** با استفاده از تابع عضو **read** می‌خواند. سپس خط 152 به مقایسه مقدار برگشتی توسط **getAccountNumber** از ساختار **client** با صفر می‌کند تا تعیین کند که آیا حاوی اطلاعات است یا خیر. اگر این مقدار صفر باشد، خطوط 175-174 یک پیغام خطا چاپ می‌کنند که دلالت بر خالی (تهی) بودن رکورد دارد. اگر رکورد حاوی اطلاعات باشد، خط 154 رکورد را با استفاده از تابع **outputLine** به نمایش در آورده، خط 159 مقدار تراکنشی را وارد، خطوط 171-162 موجودی جدید را محاسبه و رکورد مجدداً در فایل نوشته می‌شود. خروجی نمونه از گزینه2 در زیر آورده شده است

```
Enter account to update (1 – 100): 37
37         Barker         Doug           0.00
```

```
Enter charge (+) or payment (-): +87.99
37        Barker         Doug         87.99
```

گزینه3، تابع **newRecord** را برای افزودن یک حساب جدید به فایل فراخوانی می‌کند (خط 221-179). اگر کاربر شماره حسابی وارد کند که از قبل وجود داشته باشد، **newRecord** یک پیغام خطا مبنی بر وجود حساب به نمایش در می‌آورد (خطوط 220-219). این تابع یک حساب جدید به همان روش بکار رفته در برنامه شکل ۱۲-۱۷ اضافه می‌کند. خروجی نمونه از گزینه3 در زیر آورده شده است

```
Enter new account number (1 – 100): 22
Enter lastname, firstname, balance
? Johnston Sarah 247.45
```

گزینه4 تابع **deleteRecord** را برای حذف یک رکورد از فایل فراخوانی می‌کند (خطوط 255-224). خط 227 به کاربر اعلان می‌کند تا شماره حسابی را وارد سازد. فقط امکان حذف از میان رکوردهای موجود وجود دارد، از اینرو اگر حساب انتخاب شده تهی باشد، خط 254 یک پیغام خطا به نمایش در می‌آورد. اگر حساب موجود باشد، خطوط 249-247 آن حساب را با کپی کردن یک رکورد خالی (**blankClient**) بر روی آن مجدداً مقداردهی اولیه می‌کنند. خط 251 پیغامی به نمایش درآورده و به کاربر اطلاع می‌دهد که رکورد حذف شده است. خروجی نمونه از گزینه4 در زیر آورده شده است.

```
Enter account to delete (1 – 100): 29
Account #29 deleted
```

۱۲-۱۷ شی‌های از ورودی/خروجی

در این فصل و فصل پانزدهم به معرفی روش شی‌گرای C++ در ورودی/خروجی پرداختیم. با این همه، مثال‌های مطرح شده در اینجا متمرکز بر عملیات I/O بر روی نوع داده‌های متداول بجای شی‌های از نوع تعریف شده توسط کاربر بودند. در فصل یازدهم، نشان دادیم که چگونه شی‌ها با استفاده از سربارگذاری عملگر وارد و خارج می‌شوند. ورودی شی را با سربارگذاری عملگر << بر روی **istream** متناسب، انجام دادیم و خروجی شی را توسط سربارگذاری عملگر >> بر روی **osream** متناسب پیاده‌سازی کردیم. در هر دو مورد، فقط اعضای داده شی‌ها وارد یا خارج شدند و در هر مورد، دارای قالب‌بندی با معنی فقط بر روی شی‌ها از نوع داده انتزاعی خاص بودند. توابع عضو شی با داده شی وارد یا خارج نمی‌شوند، بجای آن، یک کپی از توابع عضو کلاس بصورت داخلی نگهداری می‌شوند و توسط تمام شی‌های کلاس به اشتراک گذاشته می‌شوند.

زمانیکه اعضای داده شی بر روی یک فایل دیسک نوشته می‌شوند، اطلاعات نوع شی از دست می‌رود، فقط بایت‌های داده و نه اطلاعات نوع را بر روی دیسک ذخیره می‌کنیم. اگر برنامه‌ای که این داده‌ها را می‌خواند از نوع شی مرتبط با داده مطلع باشد، برنامه داده‌ها را از درون شی‌های از آن نوع خواهد خواند.

از همه جالب‌تر اینجاست که شی‌های از نوع‌های متفاوت را در یک فایل ذخیره سازیم. چگونه می‌توانیم مابین آنها تمایز قائل شویم؟ مشکل اینجاست که معمولاً شی‌ها دارای فیلدهای نوع نیستند.

یک راه حل برای این مشکل می‌تواند این باشد که هر عملگر سربارگذاری شده خروجی، یک کد نوع قبل از هر مجموعه اعضای داده که نشاندهنده یک شی هستند، به نمایش در آورد (یا بنویسد). سپس وارد شدن شی همیشه با خواندن فیلد کد نوع شروع شده و با استفاده از یک عبارت **switch** می‌توان تابع سربارگذاری شده مناسب را فراخوانی کرد. اگرچه این روش فاقد ظرافت برنامه‌نویسی چند ریختی است، اما یک مکانیزم عملی برای حفظ شی‌ها در فایل‌ها و بازیابی آنها هنگام نیاز است.

فصل هیجدهم

کلاس string و پردازش رشته

اهداف

- استفاده از کلاس string موجود در کتابخانه استاندارد C++.
- تخصیص دادن، به هم پیوستن، مقایسه، جستجو و عوض کردن رشته‌ها.
- تعیین خصوصیات string.
- یافتن، جایگزین کردن و وارد ساختن کاراکترها در یک رشته.
- تبدیل رشته‌ها به رشته‌های سبک C و برعکس.
- استفاده از تکرار شونده‌های string.
- انجام عملیات ورودی و خروجی رشته‌ها در حافظه.

رئوس مطالب

۱۸-۱ مقدمه

الگوی کلاس **basic_string** در C++ سرمشق انواع عملیات بکار رفته بر روی رشته‌ها همانند کپی، جستجو و موارد دیگر است. تعریف الگو و تمام امکانات پشتیبانی شده در فضای‌نامی **std** تعریف شده‌اند، که با عبارت **typedef** به حساب می‌آیند

```
typedef basic_string< char > string;
```

که نام جانشین **string** را برای **basic_string<char>** ایجاد می‌کند. همچنین یک **typedef** برای نوع **wchar_t** وجود دارد. نوع **wchar_t** اقدام به ذخیره کاراکترها می‌کند (مثلاً کاراکترهای دو بایتی، چهار بایتی و غیره) تا از دیگر مجموعه‌های کاراکتری پشتیبانی شود. در این فصل انحصاراً از **string** استفاده کرده‌ایم. برای استفاده از رشته‌ها، فایل سرآیند <**string**> بکار گرفته می‌شود.

یک شی **string** می‌تواند توسط آرگومان یک سازنده همانند،

```
string text( "Hello" ); //creates string form const char *
```

مقداردهی اولیه شود، که یک رشته حاوی کاراکترهای موجود در "Hello" ایجاد می‌کند، یا با دو آرگومان سازنده همانند

```
string name( 8,'x' ); //string of 8 'x' charcters
```

که رشته‌ای حاوی هشت کاراکتر 'x' تولید می‌کند. همچنین کلاس **string** دارای سازنده پیش‌فرض (که یک رشته تهی ایجاد می‌کند) و سازنده کپی کننده است. یک *رشته تهی* (*empty string*) رشته‌ای است که حاوی هیچ کاراکتری نمی‌باشد.

همچنین می‌توان رشته‌ها را از طریق گرامر ساخت متناوب یا جایگزین در تعریف یک رشته مقداردهی اولیه کرد همانند

```
string month = "March"; //same as: string month( "March" );
```

بخاطر داشته باشید که عملگر = در عبارت قبلی یک عملگر تخصیص نمی‌باشد، بلکه فراخوانی ضمنی سازنده کلاس **string** است که عمل تبدیل را انجام می‌دهد. دقت نماید که کلاس **string** تبدیلی از **int** یا **char** به رشته در تعریف یک رشته انجام نمی‌دهد. برای مثال نتیجه تعاریف زیر

```
sring error1 = 'c';
string error2 = ( 'u' );
string error3 = 22;
string error4( 8 );
```

خطاهای نحوی است. توجه کنید که تخصیص یک کاراکتر به یک شی **string** در یک عبارت تخصیص همانند زیر مجاز است

```
string1 = 'n';
```

برخلاف رشته‌های * **char** در سبک C، رشته‌ها ضرورتاً با **null** خاتمه پیدا نمی‌کنند. طول یک رشته را می‌توان با توابع عضو **length** و **size** بازیابی کرد. عملگر شاخص، **[]**، می‌تواند به همراه رشته‌ها به منظور دسترسی و اصلاح کاراکترهای مجزا بکار گرفته شود. همانند رشته‌های سبک C، رشته‌ها در اولین شاخص مقدار صفر و آخرین شاخص مقدار **length () - 1** دارند.

اکثر توابع عضو **string** آرگومان‌های بعنوان شاخص موقعیت شروع و تعداد کاراکترهای که بر روی آنها کار خواهد شد دریافت می‌کنند.

عملگر << برای پشتیبانی از رشته‌ها سرسربارگذاری شده است. عبارت

```
string stringObject;
cin >> stringObject;
```

رشته‌ای از یک دستگاه ورودی می‌خواند. ورودی توسط کاراکترهای *white-space* تعیین حدود می‌شود. زمانیکه با چنین کاراکتری مواجه شود، عملیات ورودی خاتمه می‌یابد. همچنین تابع **getline** برای رشته سرسربارگذاری شده است. عبارت

```
string string1;
getline( cin, string1);
```

رشته را از صفحه‌کلید بدرون **string1** می‌خواند. حدود ورودی توسط خط جدید ('**\n**') مشخص می‌شود، از اینرو **getLine** می‌تواند یک خط از متن را بدرون یک شی **string** بخواند.

۱۸-۲ تخصیص و به هم پیوستن رشته‌ها

برنامه شکل ۱-۱۸ به بررسی تخصیص و به هم پیوستن رشته‌ها پرداخته است. خط 7 سرآیند <**string**> را برای کلاس رشته بکار برده است. رشته‌های **string1**، **string2** و **string3** در خطوط 14-12 ایجاد می‌شوند. خط 16 مقدار **string1** را به **string2** تخصیص می‌دهد. پس از انجام تخصیص **string2** کپی از

string1 خواهد بود. خط 17 از تابع عضو **assign** برای کپی کردن **string1** به **string3** استفاده کرده است. یکی کپی مجزا ایجاد می‌گردد (یعنی **string1** و **string3** شی‌های مستقلی هستند). همچنین کلاس رشته دارای نسخه سرسربارگذاری شده‌ای از تابع عضو **assign** است که به تعداد مشخص شده از کاراکترها را کپی می‌کند، بصورت زیر

```
targetString.assign( sourceString, start, numberOfCharacters );
```

در عبارت فوق، **sourceString** رشته‌ای است که کپی خواهد شد، **start** شاخص شروع بوده و **numberOfCharacters** تعداد کاراکترهای است که کپی می‌شود.

خط **22** از عملگر شاخص برای تخصیص **'r'** به **string3[2]** و تخصیص **'r'** به **string2[0]** استفاده کرده است. سپس رشته‌ها چاپ شده‌اند.

```
1  // Fig. 18.1: Fig18_01.cpp
2  // Demonstrating string assignment and concatenation.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <string>
8  using std::string;
9
10 int main()
11 {
12    string string1( "cat" );
13    string string2;
14    string string3;
15
16    string2 = string1; // assign string1 to string2
17    string3.assign( string1 ); // assign string1 to string3
18    cout << "string1: " << string1 << "\nstring2: " << string2
19       << "\nstring3: " << string3 << "\n\n";
20
21    // modify string2 and string3
22    string2[ 0 ] = string3[ 2 ] = 'r';
23
24    cout << "After modification of string2 and string3:\n"<< "string1:"
25       << string1 << "\nstring2: " << string2 << "\nstring3: ";
26
27    // demonstrating member function at
28    for ( int i = 0; i < string3.length(); i++ )
29       cout << string3.at( i );
30
31    // declare string4 and string5
32    string string4( string1 + "apult" ); // concatenation
33    string string5;
34
35    // overloaded +=
36    string3 += "pet"; // create "carpet"
37    string1.append( "acomb" ); // create "catacomb"
38
39    // append subscript locations 4 through end of string1 to
40    // create string "comb" (string5 was initially empty)
41    string5.append( string1, 4, string1.length() - 4 );
42
43    cout << "\n\nAfter concatenation:\nstring1: " << string1
44       << "\nstring2: " << string2 << "\nstring3: " << string3
45       << "\nstring4: " << string4 << "\nstring5: " << string5 << endl;
46    return 0;
47 }  // end main
```

```
string1: cat
string2: cat
string3: cat

After modification of string2 and string3:
```

```
string1: cat
string2: rat
string3: car

After concatenation:
string1: catacomb
string2: rat
string3: carpet
string4: catapult
string5: comb
```

شکل ۱-۱۸ | تخصیص و هم پیوستن رشته‌ها.

خطوط 29-28 محتویات **string3** را یک به یک توسط تابع عضو **at** چاپ می‌کنند. تابع عضو **at** دارای قابلیت دسترسی بررسی شده است (یا محدودهٔ بررسی شده است). به این معنی که اگر از انتهای رشته عبور شود، استثنا **out_of_range** به راه می‌افتد. توجه کنید که عملگر شاخص **[]** دسترسی بررسی شده را انجام نمی‌دهد. اینکار بر روی آرایه‌ها انجام می‌شود.

رشته **string4** در خط 32 اعلان شده و با نتیجه به هم پیوستن **string1** و "apult" با استفاده از عملگر جمع سرسربارگذاری شده، +، که بر روی کلاس رشته نقش پیوند زننده را بازی می‌کند، مقداردهی اولیه شده است. در خط 36 از عملگر =+ برای پیوند زدن **string3** و "pet" استفاده شده است. خط 37 از تابع عضو **append** برای پیوند زدن **string1** و "acomb" استفاده کرده است.

خط 41 رشته "comb" را به رشته تهی **string5** متصل کرده است. این تابع عضو مبادرت به ارسال رشته **string1** به منظور بازیابی کاراکترها از، شاخص شروع از **(4)string** و تعداد کاراکترها برای متصل شدن کرده است (مقدار برگشتی توسط **(string1.Length() - 4**)

۱۸-۳ مقایسه رشته‌ها

کلاس رشته دارای چندین تابع عضو برای مقایسه کردن رشته‌ها است. برنامه شکل ۲-۱۸ به بررسی قابلیت‌های مقایسه‌ای در کلاس رشته پرداخته است. برنامه در خطوط 15-12 مبادرت به اعلان چهار رشته کرده است و هر رشته را چاپ می‌کند (خطوط 18-17). شرط موجود در خط 21 به مقایسه رشته **string1** با **string4** به لحاظ تساوی می‌پردازد و اینکار را توسط عملگر سرسربارگذاری شده تساوی یا برابری انجام می‌دهد. اگر شرط برقرار باشد، "string1== string4" چاپ می‌شود. اگر شرط برقرار نباشد، شرط موجود در خط 25 تست می‌شود. تمام عملگرهای سرسربارگذاری شده در کلاس **string** در این برنامه بکار گرفته شده‌اند (**!=**, >، =< و =>) اما به توضیح خط به خط آنها نپرداخته‌ایم.

```
1  // Fig. 18.2: Fig18_02.cpp
2  // Demonstrating string comparison capabilities.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <string>
8  using std::string;
9
10 int main()
11 {
```

```
12    string string1( "Testing the comparison functions." );
13    string string2( "Hello" );
14    string string3( "stinger" );
15    string string4( string2 );
16
17    cout << "string1: " << string1 << "\nstring2: " << string2
18       << "\nstring3: " << string3 << "\nstring4:" << string4 << "\n\n";
19
20    // comparing string1 and string4
21    if ( string1 == string4 )
22       cout << "string1 == string4\n";
23    else // string1 != string4
24    {
25       if ( string1 > string4 )
26          cout << "string1 > string4\n";
27       else // string1 < string4
28          cout << "string1 < string4\n";
29    } // end else
30
31    // comparing string1 and string2
32    int result = string1.compare( string2 );
33
34    if ( result == 0 )
35       cout << "string1.compare( string2 ) == 0\n";
36    else // result != 0
37    {
38       if ( result > 0 )
39          cout << "string1.compare( string2 ) > 0\n";
40       else // result < 0
41          cout << "string1.compare( string2 ) < 0\n";
42    } // end else
43
44    // comparing string1 (elements 2-5) and string3 (elements 0-5)
45    result = string1.compare( 2, 5, string3, 0, 5 );
46
47    if ( result == 0 )
48       cout << "string1.compare( 2, 5, string3, 0, 5 ) == 0\n";
49    else // result != 0
50    {
51       if ( result > 0 )
52          cout << "string1.compare( 2, 5, string3, 0, 5 ) > 0\n";
53       else // result < 0
54          cout << "string1.compare( 2, 5, string3, 0, 5 ) < 0\n";
55    } // end else
56
57    // comparing string2 and string4
58    result = string4.compare( 0, string2.length(), string2 );
59
60    if ( result == 0 )
61       cout << "string4.compare( 0, string2.length(), "
62          << "string2 ) == 0" << endl;
63    else // result != 0
64    {
65       if ( result > 0 )
66          cout << "string4.compare( 0, string2.length(), "
67             << "string2 ) > 0" << endl;
68       else // result < 0
69          cout << "string4.compare( 0, string2.length(), "
70             << "string2 ) < 0" << endl;
71    } // end else
72
73    // comparing string2 and string4
74    result = string2.compare( 0, 3, string4 );
75
76    if ( result == 0 )
77       cout << "string2.compare( 0, 3, string4 ) == 0" << endl;
78    else // result != 0
79    {
80       if ( result > 0 )
81          cout << "string2.compare( 0, 3, string4 ) > 0" << endl;
```

```
82       else // result < 0
83          cout << "string2.compare( 0, 3, string4 ) < 0" << endl;
84    } // end else
85
86    return 0;
87 } // end main
```

```
string1: Testing the comparison functions.
string2: Hello
string3: stinger
string4: Hello

string1> string4
string1.compare( string2 ) > 0
string1.compare( 2, 5, string3, 0, 5 ) == 0
string4.compare( 0, string2.length(), string2 ) == 0
string2.compare( 0, 3, string4 ) < 0
string1.compare( string2 ) >0
```

شکل ۲-۱۸ | مقایسه رشته‌ها.

خط 32 از تابع عضو **compare** برای مقایسه **string1** با **string2** استفاده کرده است. اگر رشته‌ها با هم برابر باشند، متغیر **result** با صفر مقداردهی می‌شود. اگر **string1** بزرگتر از **string2** باشد با یک عدد مثبت، یا اگر **string1** کوچکتر از **string2** باشد با یک عدد منفی مقداردهی می‌شود. بدلیل اینکه رشته‌ای که با حرف **'T'** شروع می‌شود به لحاظ لغت‌نویسی بزرگتر از رشته‌ای است که با **'H'** شروع می‌شود، پس مقداری بزرگتر از صفر به **result** تخصیص می‌یابد (خروجی هم بر این نکته تاکید دارد).

خط 45 از نسخه سربارگذاری شده تابع عضو **compare** برای مقایسه بخش‌های از **string1** و **string3** استفاده کرده است. دو آرگومان اول (2 **و** 5) مشخص کننده شاخص شروع و طول بخشی از **string1** یعنی "string" برای مقایسه با **string3** هستند. آرگومان سوم، رشته مقایسه شونده است. دو آرگومان آخر (0 و 5) شاخص شروع و طول بخشی از رشته هستند که مورد مقایسه قرار خواهند گرفت.

در صورت برابری، مقدار صفر به **result**، در صورت بزرگتر بودن **string1** از **string3** مقدار مثبت و در صورت کوچکتر بودن **string1** از **string3** مقدار منفی به **result** تخصیص می‌یابد. بدلیل اینکه در این برنامه دو رشته مقایسه شده با هم یکسان هستند، **result** با صفر مقداردهی شده است.

خط 58 از نسخه سربارگذاری شده دیگری از تابع **compare** برای مقایسه **string4** و **string2** استفاده کرده است. دو آرگومان اول همان هستند، شاخص شروع و طول. آرگومان آخر، رشته مقایسه شونده است. مقدار برگشتی هم همان است، صفر برای تساوی، مقدار مثبت اگر **string4** بزرگتر از **string2** باشد یا مقدار منفی اگر **string4** کوچکتر از **string2** باشد. چون دو بخش مقایسه شده در این برنامه با هم یکسان هستند، **result** با صفر مقداردهی شده است.

خط 74 با فراخوانی تابع عضو **compare** مبادرت به مقایسه سه کاراکتر اول در رشته **string2** با **string4** کرده است. چون "Hel" کوچکتر از "Hello" است، مقداری کوچکتر از صفر برگشت داده شده است.

۱۸-۴ زیر رشته‌ها

کلاس رشته دارای تابع عضو **substr** برای بازیابی یک زیررشته از یک رشته است. نتیجه یک رشته جدید است که از رشته منبع کپی شده است. برنامه شکل ۳-۱۸ به بررسی **substr** پرداخته است.

```
1  // Fig. 18.3: Fig18_03.cpp
2  // Demonstrating string member function substr.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <string>
8  using std::string;
9
10 int main()
11 {
12    string string1( "The airplane landed on time." );
13
14    // retrieve substring "plane" which
15    // begins at subscript 7 and consists of 5 elements
16    cout << string1.substr( 7, 5 ) << endl;
17    return 0;
18 } // end main
 plane
```

شکل ۳-۱۸ | بررسی تابع عضو substr.

برنامه در خط 12 یک رشته اعلان و مقداردهی اولیه کرده است. در خط 16 از تابع عضو **substr** برای بازیابی یک زیر رشته از **string1** استفاده شده است. آرگومان اول مشخص کننده شاخص آغاز از زیررشته مورد نظر، آرگومان دوم مشخص کننده طول زیررشته است.

۱۸-۵ عوض کردن رشته‌ها

کلاس رشته دارای تابع عضو **swap** برای عوض کردن رشته‌ها است. در برنامه شکل ۴-۱۸ دو رشته عوض شده‌اند. خطوط 13-12 مبادرت به اعلان و مقداردهی رشته‌های **first** و **second** کرده‌اند. سپس هر رشته چاپ شده است. خط 18 از تابع عضو **swap** برای عوض کردن مقادیر **first** و **second** استفاده کرده است. مجدداً دو رشته چاپ شده‌اند تا تاییدی برای عوض شدن جای رشته‌ها باشد. این تابع مناسب برنامه‌های است که مرتب‌سازی رشته‌ها را انجام می‌دهند.

```
1  // Fig. 18.4: Fig18_04.cpp
2  // Using the swap function to swap two strings.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <string>
8  using std::string;
9
10 int main()
11 {
12    string first( "one" );
13    string second( "two" );
14
15    // output strings
16    cout << "Before swap:\n first: " << first << "\nsecond: " << second;
17
18    first.swap( second ); // swap strings
```

```
19
20    cout << "\n\nAfter swap:\n first: " << first
21       << "\nsecond: " << second << endl;
22    return 0;
23 } // end main
```

```
Before swap:
 first: one
second: two

After swap:
 first: two
second: one
```

شکل ۴-۱۸ | استفاده از تابع swap برای عوض کردن رشته‌ها.

۶-۱۸ خصوصیات string

کلاس **string** دارای توابع عضوی برای جمع‌آوری اطلاعاتی در ارتباط با سایز، طول، ظرفیت، حداکثر طول و خصوصیات دیگر رشته است. سایز یا طول یک رشته تعداد کاراکترهای جاری ذخیره شده در رشته است. ظرفیت یا گنجایش یک رشته تعداد کاراکترهای است که می‌تواند در یک رشته بدون اخذ حافظه بیشتر ذخیره گردد. ظرفیت یک رشته بایستی حداقل برابر سایز جاری رشته باشد، اگرچه می‌تواند بزرگتر هم باشد.

ظرفیت دقیق یک رشته وابسته به پیاده‌سازی آن است. بزرگترین سایز، سایز ممکنه‌ای است که یک رشته می‌تواند داشته باشد. اگر از این مقدار تجاوز شود، استثنا **length_error** رخ می‌دهد. برنامه شکل ۵-۱۸ به بررسی توابع عضو کلاس رشته پرداخته و از این خصوصیات در آن استفاده شده است.

```
1  // Fig. 18.5: Fig18_05.cpp
2  // Demonstrating member functions related to size and capacity.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6  using std::cin;
7  using std::boolalpha;
8
9  #include <string>
10 using std::string;
11
12 void printStatistics( const string & );
13
14 int main()
15 {
16    string string1;
17
18    cout << "Statistics before input:\n" << boolalpha;
19    printStatistics( string1 );
20
21    // read in only "tomato" from "tomato soup"
22    cout << "\n\nEnter a string: ";
23    cin >> string1; // delimited by whitespace
24    cout << "The string entered was: " << string1;
25
26    cout << "\nStatistics after input:\n";
27    printStatistics( string1 );
28
29    // read in "soup"
30    cin >> string1; // delimited by whitespace
31    cout << "\n\nThe remaining string is: " << string1 << endl;
32    printStatistics( string1 );
```

```
33
34    // append 46 characters to string1
35    string1 += "1234567890abcdefghijklmnopqrstuvwxyz1234567890";
36    cout << "\n\nstring1 is now: " << string1 << endl;
37    printStatistics( string1 );
38
39    // add 10 elements to string1
40    string1.resize( string1.length() + 10 );
41    cout << "\n\nStats after resizing by (length + 10):\n";
42    printStatistics( string1 );
43
44    cout << endl;
45    return 0;
46 } // end main
47
48 // display string statistics
49 void printStatistics( const string &stringRef )
50 {
51    cout << "capacity: " << stringRef.capacity() << "\nmax size: "
52       << stringRef.max_size() << "\nsize: " << stringRef.size()
53       << "\nlength: " << stringRef.length()
54       << "\nempty: " << stringRef.empty();
55 } // end printStatistics
```

```
Statistics befor input:
capacity: 0
max size: 4294967293
size: 0
length: 0
empty: true

Enter a string: tomato soup
The string entered was: tomato
Statistics after input:
capacity: 15
max size: 4294967293
size: 6
length: 6
empty: false

The remaing string is: soup
capacity: 15
max size: 4294967293
size: 4
length: 4
empty: false

string1 is now: soup1234567890abcdefghijklmnopqrstuvwxyz1234567890
capacity: 63
max size: 4294967293
size: 50
length: 50
empty: false

Stats after resizing by (length + 10):
capacity: 63
max size: 4294967293
size: 60
length: 60
empty: false
```

شکل ۵-۱۸ | چاپ خصوصیات رشته.

برنامه رشته تهی **string1** را اعلان (خط 16) و آنرا به تابع **printStatistics** ارسال می‌کند (خط 19). این تابع در خطوط 55-49 یک مراجعه به یک رشته ثابت بعنوان آرگومان دریافت و ظرفیت (با استفاده از تابع عضو **capacity**)، حداکثر سایز (با استفاده از تابع عضو **max_size**)، سایز (با استفاده از تابع عضو **size**)،

طول (با استفاده از تابع عضو **length**) و اینکه آیا رشته تهی است یا خیر (با استفاده از تابع عضو **empty**) را چاپ می‌کند. در ابتدای فراخوانی **printStatistics** دیده می‌شود که مقادیر اولیه برای ظرفیت، سایز و طول رشته **string1** همگی صفر است.

طول و سایز صفر بر این نکته دلالت دارند که هیچ کاراکتری در رشته ذخیره نشده است. چون ظرفیت اولیه صفر است، زمانیکه کاراکترها در **string1** جای داده می‌شوند، حافظه برای تطبیق شدن با کاراکترهای جدید، اخذ می‌شود. بخاطر داشته باشد که سایز و طول همیشه با هم یکسان هستند. در این پیاده‌سازی حداکثر سایز 4294967293 است. شی **string1** یک رشته تهی است، از اینرو تابع **empty**، مقدار **true** برگشت می‌دهد.

خط 23 از طریق خط فرمان رشته‌ای را می‌خواند. در این مثال، رشته ورودی "tomato soup" است. بدلیل اینکه کاراکتر فاصله بعنوان حد اعمال می‌شود، فقط "tomato" در **string1** ذخیره می‌گردد، با این وجود، "soup" در بافر ورودی باقی می‌ماند. خط 27 تابع **printStatistics** را برای چاپ آمار متعلق به **string1** فراخوانی می‌کند. توجه کنید که در خروجی، طول برابر 6 و ظرفیت برابر 15 است.

خط 30، رشته "soup" را از بافر ورودی خوانده و آنرا در **string1** ذخیره می‌کند، بنابر این جایگزین "tomato" می‌شود. خط 32 رشته **string1** را به تابع **printStatistics** ارسال می‌کند.

خط 32 از عملگر سربارگذاری شده =+ برای پیوند دادن یک رشته 46 کاراکتری به **string1** استفاده کرده است. خط 32 رشته **string1** را به **printStatistics** ارسال می‌کند. دقت کنید که ظرفیت به 63 عنصر و طول به 50 افزایش یافته است.

خط 40 از تابع عضو **resize** برای افزایش طول **string1** به میزان 10 کاراکتر استفاده کرده است. عناصر اضافی با کاراکترهای **null** تنظیم می‌شود. دقت کنید که در خروجی ظرفیت تغییری نکرده و طول به 60 رسیده است.

۱۸-۷ یافتن رشته‌ها و کاراکترها در یک رشته

کلاس رشته دارای توابع عضو ثابت (***const***) برای یافتن زیررشته‌ها و کاراکترها در یک رشته است. برنامه شکل ۶-۱۸ به بررسی توابع با قابلیت یافتن رشته پرداخته است.

```
1  // Fig. 18.6: Fig18_06.cpp
2  // Demonstrating the string find member functions.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <string>
8  using std::string;
9
10 int main()
11 {
12    string string1( "noon is 12 pm; midnight is not." );
13    int location;
```

```
14
15    // find "is" at location 5 and 25
16    cout << "Original string:\n" << string1
17       << "\n\n(find) \"is\" was found at: " << string1.find( "is" )
18       << "\n(rfind) \"is\" was found at: " << string1.rfind( "is" );
19
20    // find 'o' at location 1
21    location = string1.find_first_of( "misop" );
22    cout << "\n\n(find_first_of) found '" << string1[ location ]
23       << "' from the group \"misop\" at: " << location;
24
25    // find 'o' at location 29
26    location = string1.find_last_of( "misop" );
27    cout << "\n\n(find_last_of) found '" << string1[ location ]
28         << "' from the group \"misop\" at: " << location;
29
30    // find '1' at location 8
31    location = string1.find_first_not_of( "noi spm" );
32    cout << "\n\n(find_first_not_of) '" << string1[ location ]
33       << "' is not contained in \"noi spm\" and was found at:"
34       << location;
35
36    // find '.' at location 12
37    location = string1.find_first_not_of( "12noi spm" );
38    cout << "\n\n(find_first_not_of) '" << string1[ location ]
39       << "' is not contained in \"12noi spm\" and was "
40       << "found at:" << location << endl;
41
42    // search for characters not in string1
43    location = string1.find_first_not_of(
44       "noon is 12 pm; midnight is not." );
45    cout << "\nfind_first_not_of(\"noon is 12 pm; midnight is not.\")"
46       << " returned: " << location << endl;
47    return 0;
48 } // end main
```

```
Original string:
noon is 12 pm; midnight is not.

(find) "is"was found at: 5
(rfind) "is"was found at: 25

(find_first_of) found 'O' from the group "misop" at: 1

(find_first_not_of) '1' is not contained in "noi spm" and was found at: 8

(find_first_not_of) '.' is not contained in "12noi spm" and was found at:
12

find_first_not_of("noon is 12 pm; midnight is not.") returned: -1
```

شکل ٦-١٨ | توابع یافتن رشته.

در خط 12 رشته **string1** اعلان و مقداردهی اولیه شده است. خط 17 مبادرت به یافتن "is" در رشته **string1** با استفاده از تابع **find** کرده است. اگر "is" پیدا شود، شاخص موقعیت شروع از آن رشته برگشت داده می‌شود. اگر رشته پیدا نشود، مقدار **string::npos** (یک ثابت استاتیک عمومی تعریف شده در کلاس **string**) برگشت داده خواهد شد. این مقدار توسط توابع مرتبط با یافتن رشته برگشت داده می‌شود تا نشان داده شود که زیررشته یا کاراکتر در رشته پیدا نشده است.

خط 18 از تابع عضو **find**، برای جستجوی پس‌گشت (یعنی از راست به چپ) رشته **string1** استفاده کرده است. اگر "is" پیدا شود، شاخص موقعیت آن برگشت داده می‌شود. اگر رشته پیدا نشود، **string::npos** برگشت داده خواهد شد.

خط 21 از تابع عضو **find_first_of** برای یافتن اولین پیشامد در رشته **string1** از هر کاراکتری در "misop" استفاده کرده است. جستجو از ابتدای **string1** شروع می‌شود. کاراکتر "o" در عنصر 1 پیدا شده است.

خط 26 از تابع عضو **find_last_of** برای یافتن آخرین پیشامد در رشته **string1** از هر کاراکتری در "misop" استفاده کرده است. جستجو از انتهای **string1** شروع می‌شود. کاراکتر 'o' در عنصر 29 پیدا شده است.

خط 31 از تابع عضو **find_first_not_of** برای یافتن اولین کاراکتر در **string1** که حاوی "noi spm" نیست، استفاده کرده است. کاراکتر '1' در عنصر 8 پیدا شده است. جستجو از ابتدا **string1** شروع می‌شود.

خط 37 از تابع **find_first_not_of** برای یافتن اولین کاراکتر که در "12noi spm" وجود ندارد، استفاده کرده است. کاراکتر '.' در عنصر 12 پیدا شده است. جستجو از انتهای رشته صورت گرفته است.

خطوط 33-43 از تابع عضو **find_first_not_of** برای یافتن اولین کاراکتری که در "noon is 12 pm; midnight is not." وجود ندارد استفاده کرده‌اند. در این مورد، رشته جستجو شده حاوی هر کاراکتر مشخص شده در آرگومان رشته است. بدلیل اینکه کاراکتری پیدا نشده است، **string::npos** برگشت داده شده است (که در این مورد، مقدار 1- دارد).

۱۸-۸ جایگزین ساختن کاراکترها در یک رشته

برنامه شکل ۷-۱۸ به بررسی توابع عضو رشته که در ارتباط با جایگزین‌سازی و پاک کردن کاراکترها هستند، پرداخته است. خطوط 17-13 رشته **string1** را اعلان و مقداردهی اولیه کرده‌اند. خط 23 از تابع عضو **erase** برای پاک کردن هر چیزی از موقعیت 62 تا انتهای رشته **string1** استفاده کرده است.

خطوط 36-29 از **find** برای یافتن هر پیشامدی از کاراکتر فاصله استفاده کرده‌اند. سپس هر فاصله با یک نقطه توسط تابع **replace** جایگزین شده است. تابع **replace** سه آرگومان دریافت می‌کند: شاخص کاراکتر در رشته که بایستی جایگزینی از آنجا آغاز شود، تعداد کاراکترها برای جایگزینی و رشته جایگزین. تابع عضو **find** در زمانیکه کاراکتر مورد جستجو پیدا نشود، **string::npos** برگشت می‌دهد. در خط 35، عدد 1 به **position** افزوده شده تا جستجو از موقعیت کاراکتر بعدی ادامه یابد.

در خطوط 48-40 از تابع **find** برای یافتن هر نقطه و تابع سربارگذاری شده **replace** برای جایگزین ساختن هر نقطه و کاراکتر بعدی آن با دو سیمکولن استفاده شده است. آرگومان‌های ارسالی به این نسخه از تابع **replace** عبارتند از شاخص عنصر در محلی که عملیات جایگزینی آغاز می‌شود، تعداد کاراکترهای جایگزین شونده، کاراکتر جایگزین شونده، عنصر در رشته کاراکتری در محلی که زیررشته جایگزین شروع شده و تعداد کاراکترها در رشته کاراکتری جایگزین شونده، برای استفاده.

```
1  // Fig. 18.7: Fig18_07.cpp
2  // Demonstrating string member functions erase and replace.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <string>
8  using std::string;
9
10 int main()
11 {
12    // compiler concatenates all parts into one string
13    string string1( "The values in any left subtree"
14       "\nare less than the value in the"
15       "\nparent node and the values in"
16       "\nany right subtree are greater"
17       "\nthan the value in the parent node" );
18
19    cout << "Original string:\n" << string1 << endl << endl;
20
21    // remove all characters from (and including) location 62
22    // through the end of string1
23    string1.erase( 62 );
24
25    // output new string
26    cout << "Original string after erase:\n" << string1
27       << "\n\nAfter first replacement:\n";
28
29    int position = string1.find( " " ); // find first space
30
31    // replace all spaces with period
32    while ( position != string::npos )
33    {
34       string1.replace( position, 1, "." );
35       position = string1.find( " ", position + 1 );
36    } // end while
37
38    cout << string1 << "\n\nAfter second replacement:\n";
39
40    position = string1.find( "." ); // find first period
41
42    // replace all periods with two semicolons
43    // NOTE: this will overwrite characters
44    while ( position != string::npos )
45    {
46       string1.replace( position, 2, "xxxxx;;yyy", 5, 2 );
47       position = string1.find( ".", position + 1 );
48    } // end while
49
50    cout << string1 << endl;
51    return 0;
52 } // end main
```

```
Original string:
The values in any left subtree
are less than the value in the
parent node and the values in
any right subtree are greater
than the value in the parent node
```

```
Original string after erase:
The values in any left subtree
are less than the value in the

After first replacement:
The.values.in.any.left.subtree
are.less.than.the.value.in.the

After second replacement:
The;;alues;;n;;ny;;eft;;ubtree
are;;ess;;han;;he;;alue;;n;;he
```

شکل ۷-۱۸ | بررسی توابع erase و replace.

۹-۱۸ درج کاراکترها در یک رشته

کلاس رشته دارای توابع عضوی برای درج کاراکتر در یک رشته است. برنامه شکل ۸-۱۸ به بررسی قابلیت‌های درج کلاس رشته پرداخته است.

برنامه مبادرت به اعلان، مقداردهی اولیه سپس چاپ رشته‌های **string1**، **string2**، **string3** و **string4** کرده است. خط 22 از تابع عضو **insert** برای درج محتویات **string2** قبل از عنصر 10 در رشته **string1** استفاده کرده است.

خط 25 با استفاده از تابع **insert**، مبادرت به درج **string4** قبل از عنصر 3 رشته **string3** کرده است. دو آرگومان آخر مشخص کننده عنصر شروع و پایان **string4** هستند که بایستی درج شوند. استفاده از **string::npos** سبب می‌شود تا کل رشته درج شود.

```
1  // Fig. 18.8: Fig18_08.cpp
2  // Demonstrating class string insert member functions.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <string>
8  using std::string;
9
10 int main()
11 {
12    string string1( "beginning end" );
13    string string2( "middle " );
14    string string3( "12345678" );
15    string string4( "xx" );
16
17    cout << "Initial strings:\nstring1: " << string1
18       << "\nstring2: " << string2 << "\nstring3: " << string3
19       << "\nstring4: " << string4 << "\n\n";
20
21    // insert "middle" at location 10 in string1
22    string1.insert( 10, string2 );
23
24    // insert "xx" at location 3 in string3
25    string3.insert( 3, string4, 0, string::npos );
26
27    cout << "Strings after insert:\nstring1: " << string1
28       << "\nstring2: " << string2 << "\nstring3: " << string3
29       << "\nstring4: " << string4 << endl;
30    return 0;
31 } // end main
```

```
Initial strings:
```

```
string1: beginning end
string2: middle
string3: 12345678
string4: xx

Strings after insert:
string1: beginning middle end
string2: middle
string3: 123xx45678
string4: xx
```

شکل ۸-۱۸ | بررسی تابع عضو insert.

۱۰-۱۸ تبدیل به سبک رشته‌های C

کلاس رشته دارای توابع عضوی است که می‌توانند شی‌های کلاس رشته را تبدیل به رشته‌های مبتنی بر اشاره‌گر سبک C (یعنی C-style) کنند. همانطوری که قبلاً هم گفته شده، برخلاف رشته‌های مبتنی بر اشاره‌گر، رشته‌ها ضرورتاً با **null** خاتمه پیدا نمی‌کنند. این توابع تبدیل کننده زمانی سودمند هستند که تابعی یک رشته مبتنی بر اشاره‌گر بعنوان آرگومان دریافت کند. برنامه شکل ۹-۱۸ به بررسی تبدیل رشته‌ها به رشته‌های مبتنی بر اشاره‌گر پرداخته است.

برنامه مبادرت به اعلان یک رشته، یک **int** و دو اشاره‌گر **char** کرده است (خطوط 15-12). رشته **string1** با "STRINGS"، **ptr1** با 0 و **length** با طول **string1** مقداردهی اولیه شده است. حافظه به میزان کافی برای نگهداری یک رشته مبتنی بر اشاره‌گر معادل رشته **string1** بصورت دینامیکی اخذ شده و به اشاره‌گر **ptr2** از نوع **char** الصاق می‌شود.

خط 18 از تابع عضو **copy** برای کپی شی **string1** به آرایه **char** مورد اشاره **ptr2** استفاده کرده است.

خط 19 بصورت غیراتوماتیک یک کاراکتر خاتمه دهنده **null** در آرایه مورد اشاره **ptr2** قرار می‌دهد.

خط 23 از تابع **c_str** برای کپی شی **string1** و افزودن اتوماتیک کاراکتر خاتمه دهنده **null** استفاده کرده است. این تابع یک **const char *** برگشت می‌دهد که توسط عملگر درج استریم چاپ می‌شود.

خط 29 مبادرت به تخصیص **const char * ptr1** به اشاره‌گر برگشتی توسط تابع عضو **data** می‌کند. این تابع عضو یک کاراکتر غیر **null** خاتمه دهنده آرایه کاراکتری سبک C برگشت می‌دهد. دقت کنید که در این مثال تغییری در رشته **string1** بوجود نیاورده‌ایم.

اگر رشته **string1** دچار تغییر شود (برای مثال حافظه دینامیکی رشته آدرس خود را به علت فراخوانی یک تابع عضو همانند **string1.insert(0, "abcd");** تغییر دهد)، اشاره‌گر **ptr1** نامعتبر خواهد شد، که می‌تواند نتایج غیرقابل پیش‌بینی بوجود آورد.

خطوط 33-32 از محاسبه اشاره‌گر برای چاپ آرایه کاراکتری مورد اشاره توسط **ptr1** استفاده کرده‌اند.

در خطوط 36-35 رشته سبک C مورد اشاره توسط **ptr2** چاپ شده و حافظه اخذ شده حذف می‌شود تا از فقدان حافظه جلوگیری گردد.

```
1  // Fig. 18.9: Fig18_09.cpp
```

```
2  // Converting to C-style strings.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <string>
8  using std::string;
9
10 int main()
11 {
12    string string1( "STRINGS" ); // string constructor with char* arg
13    const char *ptr1 = 0; // initialize *ptr1
14    int length = string1.length();
15    char *ptr2 = new char[ length + 1 ]; // including null
16
17    // copy characters from string1 into allocated memory
18    string1.copy( ptr2, length, 0 ); // copy string1 to ptr2 char*
19    ptr2[ length ] = '\0'; // add null terminator
20
21    cout << "string string1 is " << string1
22       << "\nstring1 converted to a C-Style string is "
23       << string1.c_str() << "\nptr1 is ";
24
25    // Assign to pointer ptr1 the const char * returned by
26    // function data(). NOTE: this is a potentially dangerous
27    // assignment. If string1 is modified, pointer ptr1 can
28    // become invalid.
29    ptr1 = string1.data();
30
31    // output each character using pointer
32    for ( int i = 0; i < length; i++ )
33       cout << *( ptr1 + i ); // use pointer arithmetic
34
35    cout << "\nptr2 is " << ptr2 << endl;
36    delete [] ptr2; // reclaim dynamically allocated memory
37    return 0;
38 } // end main
```

```
string string1 is STRINGS
string1 converted to a C-Style string is STRINGS
ptr1 is STRINGS
ptr2 is STRINGS
```

شکل ۹-۱۸ | تبدیل رشته به رشته‌ها و آرایه‌های کاراکتری سبک C.

۱۱-۱۸ تکرار شونده‌ها

کلاس رشته دارای *تکرار شونده‌های* (*iterators*) برای پیمایش به سمت جلو و عقب در میان رشته‌ها است. تکرار شونده‌ها دسترسی به کاراکترهای مجزا را با گرامری که شبیه عملیات اشاره‌گر است، فراهم می‌آورند. تکرار شونده‌ها بررسی محدودهٔ را انجام نمی‌دهند. توجه کنید که در این بخش مثال‌های مطرح شده فقط جنبه آشنا کردن شما با تکرار شونده‌ها را دارند. برنامه شکل ۱۰-۱۸ به بررسی تکرار شونده‌ها پرداخته است.

```
1  // Fig. 18.10: Fig18_10.cpp
2  // Using an iterator to output a string.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <string>
8  using std::string;
9
10 int main()
11 {
```

```
12    string string1( "Testing iterators" );
13    string::const_iterator iterator1 = string1.begin();
14
15    cout << "string1 = " << string1
16       << "\n(Using iterator iterator1) string1 is: ";
17
18    // iterate through string
19    while ( iterator1 != string1.end() )
20    {
21       cout << *iterator1; // dereference iterator to get char
22       iterator1++; // advance iterator to next char
23    } // end while
24
25    cout << endl;
26    return 0;
27 } // end main
```

```
string1 = Testing iterators
(Using iterator iterator1) string1 is: Testing iterators
```

شکل ۱۰-۱۸ | استفاده از تکرار شونده‌ها در چاپ رشته.

خطوط 13-12 رشته **string1** و تکرار شونده **string::const_iterator iterator1** را اعلان کرده‌اند. **const_iterator** تکرار شونده‌ای است که نمی‌تواند رشته را تغییر دهد. تکرار شونده **iterator1** با ابتدای **string1** توسط تابع عضو **begin** مقداردهی اولیه شده است.

دو نسخه از تابع **begin** وجود دارد، یکی که یک تکرار شونده برای حرکت در میان یک رشته غیرثابت برگشت می‌دهد و یک نسخه ثابت که یک **const_iterator** برای حرکت در میان یک رشته ثابت برگشت می‌دهد. خط 15 رشته **string1** را چاپ می‌کند.

خطوط 23-19 با استفاده از **iterator1** در میان رشته **string1** حرکت می‌کنند. تابع عضو **end** یک تکرار شونده (یا یک **const_iterator**) برای موقعیت قبل از آخرین عنصر رشته **string1** برگشت می‌دهد.

کلاس **string** دارای توابع عضو **read** و **rbegin** برای دستیابی به کاراکترهای مجزای رشته بصورت معکوس از انتها به سمت ابتدای رشته است. توابع عضو **rend** و **rbegin** می‌توانند **reverse_iterators** و **const_reverse_iterators** برگشت دهند.

۱۲-۱۸ پردازش استریم رشته

علاوه بر استریم استاندارد I/O و استریم فایل I/O، استریم I/O در C++ حاوی قابلیت‌های برای ورودی از و خروجی به رشته‌های موجود در حافظه است. غالباً از این قابلیت‌ها بعنوان I/O حافظه یا پردازش استریم رشته یاد می‌شود.

ورودی از یک رشته توسط کلاس **istringstream** و خروجی به رشته توسط کلاس **ostringstream** پشتیبانی می‌شود. اسامی کلاس **istringstream** و **ostringstream** در واقع اسامی دیگری هستند که توسط *typedef**های* زیر تعریف شده‌اند

```
typedef basic_istringstream< char > istringstream;
typedef basic_ostringstream< char > ostringstream;
```

الگوهای کلاس **basic_istringstream** و **basic_ostringstream** همان قابلیت‌های کلاس **istream** و **ostream** را به همراه تعدادی توابع عضو خاص کار با حافظه هستند. برنامه‌های که از قالب‌بندی حافظه استفاده می‌کنند بایستی حاوی فایل سرآیند <**ssream**> و <**iostream**> باشند.

یکی از کاربردهای این تکنیک‌ها، اعتبارسنجی داده‌ها می‌باشد. برنامه می‌تواند کل یک خط را در یک زمان از استریم ورودی بدرون یک رشته بخواند، سپس یک روتین اعتبارسنجی می‌تواند بدقت محتویات رشته را بررسی کرده و در صورت نیاز، داده را اصلاح (یا تعمیر) نماید. سپس برنامه می‌تواند به دریافت رشته ادامه دهد با اطمینان از اینکه داده ورودی در قالب صحیح وارد می‌شود.

خارج کردن (چاپ) یک رشته روش مناسبی برای بهره‌مند شدن از مزایای خروجی قالب‌بندی شده استریم‌ها در C++ است. داده موجود در یک رشته می‌تواند مهیای ویرایش گردد.

برنامه شکل ۱۱-۱۸ به بررسی یک شی **ostringstream** پرداخته است. برنامه شی **outputString** را ایجاد کرده (خط 15) و از عملگر درج استریم برای چاپ دنباله‌ای از رشته‌ها و مقادیر عددی استفاده می‌کند.

خطوط 28-27 مبادرت به چاپ رشته **string1**، رشته **string2**، رشته **string3**، **double1**، رشته **string4**، **integer**، رشته **string5** و آدرس **int** چاپ می‌کنند، که همگی **outputString** در حافظه هستند. خط 31 از عملگر درج استریم و فراخوانی **outputString.str()** برای نمایش یک کپی از رشته ایجاد شده در خطوط 28-27 کرده است. خط 34 به بررسی این مطلب پرداخته است که داده می‌تواند به رشته‌ای در حافظه الصاق شود که اینکار به آسانی و با صدور یک عملیات درج استریم دیگر به **outputString** صورت می‌گیرد. خطوط 36-35 رشته **outputString** را پس از الصاق کاراکترهای اضافی، چاپ می‌کنند.

```
1  // Fig. 18.11: Fig18_11.cpp
2  // Using a dynamically allocated ostringstream object.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <string>
8  using std::string;
9
10 #include <sstream> // header file for string stream processing
11 using std::ostringstream; // stream insertion operators
12
13 int main()
14 {
15    ostringstream outputString; // create ostringstream instance
16
17    string string1( "Output of several data types " );
18    string string2( "to an ostringstream object:" );
19    string string3( "\n        double: " );
20    string string4( "\n           int: " );
21    string string5( "\naddress of int: " );
22
23    double double1 = 123.4567;
```

```
24    int integer = 22;
25
26    // output strings, double and int to ostringstream outputString
27    outputString << string1 << string2 << string3 << double1
28       << string4 << integer << string5 << &integer;
29
30    // call str to obtain string contents of the ostringstream
31    cout << "outputString contains:\n" << outputString.str();
32
33    // add additional characters and call str to output string
34    outputString << "\nmore characters added";
35    cout << "\n\nafter additional stream insertions,\n"
36       << "outputString contains:\n" << outputString.str() << endl;
37    return 0;
38 } // end main
```

```
outputString contains:
Output of several data types to an ostringstream object:
        double: 123.457
           int: 22
address of int: 0012F540

after additional stream insertions,
outputString contains:
Output of several data types to an ostringstream object:
        double: 123.457
           int: 22
address of int: 0012F540
more characters added
```

شکل ۱۱-۱۸ | استفاده از شی ostringstream اخذ شده به روش دینامیکی.

یک شی **istringstream** داده را از یک رشته موجود وارد متغیرهای برنامه می‌کند. داده‌های ذخیره شده در یک شی **istringstream** نقش کاراکتر دارند. ورودی از شی **istringstream** همانند ورودی از هر فایلی عمل می‌کند. انتهای رشته توسط شی **istringstream** همانند انتهای فایل تفسیر می‌شود.

برنامه شکل ۱۲-۱۸ به بررسی ورودی از یک شی **istringstream** پرداخته است. خطوط 16-15 یک رشته **input** حاوی داده و یک شی **inpuString** ایجاد شده با داده موجود در رشته **input** ایجاد می‌کنند. رشته **input** حاوی داده زیر است.

```
Input test 123 4.7 A
```

که به هنگام خوانده شدن بعنوان ورودی به برنامه، متشکل از دو رشته ("test" و "Input")، یک (123) **int**، یک **double**(4.7) و یک **char** ('A') است. این کاراکترها به متغیرهای **string1**، **string2**، **integer**، **double1** و **character** در خط 23 انتقال داده می‌شوند.

```
1  // Fig. 18.12: Fig18_12.cpp
2  // Demonstrating input from an istringstream object.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <string>
8  using std::string;
9
10 #include <sstream>
11 using std::istringstream;
12
13 int main()
14 {
15    string input( "Input test 123 4.7 A" );
16    istringstream inputString( input );
17    string string1;
```

```
18    string string2;
19    int integer;
20    double double1;
21    char character;
22
23    inputString >> string1 >> string2 >> integer >> double1 >> character;
24
25    cout << "The following items were extracted\n"
26       << "from the istringstream object:" << "\nstring: " << string1
27       << "\nstring: " << string2 << "\n   int: " << integer
28       << "\ndouble: " << double1 << "\n  char: " << character;
29
30    // attempt to read from empty stream
31    long value;
32    inputString >> value;
33
34    // test stream results
35    if ( inputString.good() )
36       cout << "\n\nlong value is: " << value << endl;
37    else
38       cout << "\n\ninputString is empty" << endl;
39
40    return 0;
41 } // end main
```

```
The follwing items were extracted
from the istringstream object:
string: Input
string: test
   int: 123
double: 4.7
  char: A

inputString is empty
```

شکل ۱۲-۱۸ | توصیف عملکرد ورودی از طریق شی istringstream.

سپس داده‌ها توسط خطوط 28-25 خارج می‌شوند. برنامه مبادرت به خواندن مجدد **inputString** در خط 32 می‌کند. شرط **if** در خط 35 از تابع **good** برای تست اینکه آیا داده‌ای باقی مانده است، استفاده می‌کند. چون هیچ داده‌ای باقی نمانده است، تابع مقدار **false** برگشت داده و بخش **else** از عبارت **if...else** اجرا می‌شود.

فصل نوزدهم

برنامه‌نویسی وب

اهداف

- پروتکل CGI.
- سرآیندهای HTTP و HTT.
- عملکرد سرویس‌دهنده وب.
- سرویس‌دهنده Apache HTTP.
- تقاضای مستندات از سرویس‌دهنده وب.
- پیاده‌سازی اسکریپت‌های CGI.
- ارسال ورودی به اسکریپت‌های CGI توسط فرم‌های XHTML.

رئوس مطالب

۱۹-۱ مقدمه

با ظهور و ورود *وب گسترده جهانی* (WWW). اینترنت محبوبیت باور نکردنی پیدا کرد. این حجم عظیم تقاضای کاربران برای بدست آوردن اطلاعات به سمت وب سایت‌ها هدایت گردید. آشکار شد که میزان تعامل مابین کاربر و وب سایت از اهمیت خاصی برخوردار است. قدرت وب نه تنها در ارائه مطالب به کاربران است، بلکه در واکنش و پاسخ دادن به تقاضاهای کاربران نیز می‌باشد و از اینرو محتویات وب حالت دینامیکی دارند.

در این فصل، در ارتباط با نرم‌افزار خاصی بنام *سرویس‌دهنده وب* (web server) صحبت خواهیم کرد که به تقاضای *سرویس‌گیرنده* (*client*)، مثلاً مرورگر وب، با فراهم آوردن منابع (مثلاً مستندات XHTML) و نمایش آنها برای سرویس‌گیرنده، ازخود واکنش نشان می‌دهد. برای مثال زمانیکه کاربر یک آدرس URL(Uniform Resource Locator) همانند www.detial.com را در یک مرورگر وب وارد می‌کند، کاربر مستند خاصی را از یک سرویس‌دهنده وب تقاضا می‌کند. سرویس‌دهنده وب مبادرت به نگاشت URL با فایلی بر روی سرویس‌دهنده (یا فایلی بر روی شبکه سرویس‌دهنده) کرده و مستند مورد تقاضا را به سرویس‌گیرنده برگشت می‌دهد. در مدت زمان این تعامل، سرویس‌دهنده وب و سرویس‌گیرنده از طریق پروتکل HyperText Transfer Protocol (HTTP)، که یک پروتکل مستقل از پلات‌فرم (platform) است با یکدیگر در ارتباط بوده و با استفاده از این پروتکل مبادرت به انتقال تقاضاها و فایلها بر روی اینترنت می‌کنند (یعنی مابین سرویس‌دهنده‌های وب و مرورگرهای وب).

سرویس‌دهنده وبی که در اینجا معرفی می‌کنیم سرویس‌دهنده Apache HTTP است. برای نشان دادن نتایج کار را از مرورگر وب Internet Explorer شرکت Microsoft استفاده کرده‌ایم.

۲-۱۹ انواع تقاضاهای HTTP

پروتکل HTTP در برگیرنده چندین نوع از تقاضا است که با نام متدهای تقاضا شناخته می‌شوند. هر کدامیک نحوه درخواست سرویس‌گیرنده از سرویس‌دهنده را مشخص می‌کنند. دو متد متداول در این زمینه عبارتند از get و post. این تقاضاها مبادرت به بازیابی و ارسال داده به سرویس‌گیرنده و از سرویس‌دهنده وب به سرویس‌گیرنده می‌کنند. فرم یک عنصر XHTML بوده و می‌تواند حاوی فیلدهای متنی، دکمه‌های رادیوئی، جعبه‌چک‌ها و سایر کامپونت‌های گرافیکی واسط کاربر (GUI) باشد که به کاربر امکان می‌دهد تا داده‌های خود را وارد یک صفحه وب نماید. البته فرم‌ها می‌توانند در برگیرنده فیلدهای پنهان هم باشند. از یک تقاضای get برای ارسال داده به سرویس‌دهنده استفاده می‌شود. همچنین از post برای ارسال داده به سرویس‌دهنده استفاده می‌شود. متد get ، داده را بعنوان بخشی از URL ارسال می‌کند، همانند `www.searchsomething.com/search?query=`*`userquery`* که حاوی ورودی کاربر است. برای مثال، اگر کاربر جستجوی بر روی "Massachusetts" انجام دهد، بخش انتهائی URL عبارت `?query=Massachusetts` خواهد بود. در حالیکه متد *get* رشته پرس‌وجو (*query string*) را با تعدادی از کارکترهای از پیش تعریف شده محدود می‌کند(رشته پرس‌وجو در این مثال `query=Massachusetts`است). این محدودیت از سرویس‌دهنده‌ای به سرویس‌دهنده دیگر، مختلف است. اگر طول رشته پرس‌وجو از این محدوده تجاوز نماید، بایستی از یک متد *post* استفاده کرد.

مهندسی نرم‌افزار

داده ارسالی توسط متد ***post*** *جزئی از* ***URL*** *نبوده و توسط کاربر قابل رویت نمی‌باشد. غالبا فرم‌های که باید برخی از فیلدهای آن مورد تایید قرار گیرند از متد* ***post*** *استفاده می‌کنند.فیلدهای حساسی همانند کلمه رمز از این روش برای ارسال داده استفاده می‌کنند.*

غالبا یک تقاضای HTTP داده را به سمت یک سرویس‌دهنده و از طریق یک دستگیره (*handler*) که پردازش داده‌ها را انجام می‌دهد، ارسال می‌کند.

در اکثر مواقع مرورگرها مبادرت به ذخیره کردن صفحات وب بر روی دیسک محلی(*cache*) برای بار شدن سریعتر صفحات می‌کنند تا از میزان داده‌های که نیاز است تا مرورگر آنها را برداشت نماید، کاسته شود. با این وجود، در حالت کلی مرورگرها مبادرت به ذخیره کردن پاسخ‌های داده شده به

تقاضاهای post نمی‌کنند، چراکه امکان دارد تقاضاهای post بعدی حاوی همان اطلاعات قبلی نباشند. از اینرو در اکثر مواقع مرورگرهای وب پاسخ تقاضاهای *get* را ذخیره می‌کنند. جدول شکل ۱-۱۹ لیستی از انواع تقاضاها را بجز get و post ارائه کرده است. این متدها استفاده زیادی ندارند.

نوع تقاضا	توضیح
delete	همانند تقاضای است که معمولاً برای حذف فایلی از سرویس‌دهنده بکار گرفته می‌شود. غالباً این تقاضا بدلایل امنیتی قابل انجام بر روی اکثر سرویس‌دهنده‌ها نیست.
head	معمولاً از این تقاضا زمانی استفاده می‌شود که سرویس‌گیرنده می‌خواهد فقط واکنش دهنده به سرآیندها باشد، همانند نوع محتویات و طول محتویات.
options	چنین تقاضای، اطلاعاتی به سرویس‌گیرنده برگشت می‌دهد که دلالت بر گزینه‌های پشتیبانی شده HTTP از سوی سرویس‌دهنده دارند، همانند نسخه HTTP (1.1 یا 1.0) و متدهای تقاضا که سرویس‌دهنده از آنها پشتیبانی می‌کند.
put	معمولاً از چنین تقاضای برای ذخیره فایل بر روی سرویس‌دهنده استفاده می‌شود. غالباً این تقاضا بدلایل امنیتی قابل انجام بر روی اکثر سرویس‌دهنده‌ها نیست.
trace	معمولاً از چنین تقاضای برای خطایابی استفاده می‌شود.

شکل ۱-۱۹ | دیگر نوع‌های تقاضای HTTP.

۳-۱۹ معماری چند گره‌ای

سرویس‌دهنده وب بخشی از *برنامه چندگره‌ای* (*multi-tier*) است که گاهی اوقات بعنوان برنامه*n گره‌ای* نیز شناخته می‌شود. عملکرد برنامه‌های چندگره‌ای به گره‌های جداگانه (یا گروه‌های منطقی) تقسیم می‌شود. اگرچه گره‌ها می‌توانند بر روی یک کامپیوتر قرار داشته باشند، اما غالباً گره‌های برنامه‌های مبتنی بر وب بر روی کامپیوترهای جداگانه قرار داده می‌شوند. شکل ۲-۱۹ ساختار اولیه یک برنامه سه گره‌ای را نشان می‌دهد.

گره اطلاعات (با نام *گره داده* یا *گره تحتانی* نیز شناخته می‌شود) وظیفه نگهداری داده‌های برنامه را برعهده دارد. عموماً این گره، داده‌ها را در یک *سیستم مدیریت پایگاه داده رابطه‌ای* (*RDBMS*) ذخیره می‌کند. برای مثال یک فروشگاه می‌تواند دارای پایگاه داده‌ای برای ذخیره‌سازی اطلاعاتی همانند نوع

کالا، قیمت و تعداد آنها باشد. همان پایگاه داده می‌تواند حاوی اطلاعات هر مشتری همانند نام کاربر، آدرس و شماره کارت اعتباری باشد. این گره می‌تواند متشکل از چندین پایگاه داده باشد که به کمک هم اطلاعات مورد نیاز برنامه را فراهم می‌آورند.

گره میانی وظیفه پیاده‌سازی *موازنه منطقی* و *کنترل* تعامل صورت گرفته مابین برنامه سرویس‌گیرنده و داده‌های برنامه را برعهده دارد. گره میانی همانند یک میانجی مابین داده‌های موجود در گره اطلاعات و برنامه‌های سرویس‌گیرنده عمل می‌کند. گره میانی مبادرت به کنترل منطقی تقاضاهای سرویس‌گیرنده‌ها کرده و اطلاعات را از پایگاه داده بازیابی می‌کند. معمولاً برنامه‌های وب داده‌ها را بفرم مستندات HTML به سرویس‌گیرنده‌ها عرضه می‌کنند.

شکل ۲-۱۹ | معماری سه گره‌ای.

منطق موازنه در گره میانی، سبب اعمال *قوانین موازنه* می‌شود و کاری می‌کند که داده‌ها قبل از اینکه برنامه سرویس‌دهنده مبادرت به روز کردن پایگاه داده یا ارائه اطلاعات به کاربران نماید، قابل اطمینان و قابل عرضه شوند. قوانین موازنه نحوه دسترسی سرویس‌گیرنده‌ها به اطلاعات و پردازش داده‌ها را تعیین می‌کنند.

گره سرویس‌گیرنده یا *گره فوقانی*، برنامه واسط کاربر است که عموماً یک مرورگر وب می‌باشد و کاربران مستقیماً و از طریق واسط کاربر با برنامه در تعامل قرار می‌گیرند. گره سرویس‌گیرنده با گره میانی در ارتباط است تا تقاضای خود را مطرح و داده‌ها را از گره اطلاعات دریافت کند. سپس گره س‌رویس‌گیرنده داده‌های دریافتی از گره میانی را در اختیار کاربر قرار می‌دهد.

۱۹-٤ دسترسی به سرویس‌دهنده‌های وب

برای اینکه بتوان از مستندات مقیم بر روی سرویس‌دهنده‌های وب استفاده کرد، نیاز است تا با URL آنها آشنا بود. یک URL حاوی نام ماشین (به *نام میزبان* شناخته می‌شود) است که بر روی سرویس‌دهنده وب مقیم می‌باشد. کاربران می‌توانند، مستندات را از *سرویس‌دهنده‌های وب محلی* (مقیم بر روی ماشین یکی از کاربران) یا *سرویس‌دهنده‌های وب راه دور* (مقیم بر روی یکی از ماشین‌های موجود در شبکه) درخواست نمایند.

به دو روش می‌توان به سرویس‌دهنده‌های وب محلی دسترسی پیدا کرد: از طریق نام ماشین، یا از طریق localhost – نام میزبانی که اشاره به یک ماشین محلی دارد. در این فصل از localhost استفاده می‌کنیم. برای تعیین نام ماشین در ویندوز 2000، بر روی My Computer کلیک راست کرده و از

منوی ظاهر شده Properties را برای به نمایش در آمدن کادر تبادلی System Properties، انتخاب نمائید. در این کادر تبادلی، بر روی Network Identification کلیک کنید. فیلد Full Computer Name: در پنجره System Properties نام کامپیوتر را نشان می‌دهد. در ویندوز XP، منوی Start>Control Panel>Switch to Classic View> System را برای نمایش کادر تبادلی System Properties انتخاب کنید. در این کادر بر روی زبانه Computer Name کلیک نمائید.

نام دامنه نشاندهنده گروهی از میزبان‌ها در اینترنت است؛ که با نام یک میزبان (برای نمونه، **www**) و یک *دامنه سطح بالا* (TLD[1]) ترکیب می‌شوند، تا روش کاربر پسندی برای شناسائی یک سایت در اینترنت بدست آید. به هر کدامیک از این اسامی میزبان یک آدرس منحصر بفرد بنام *آدرس IP* تخصیص داده می‌شود. این آدرس‌ها بسیار شبیه آدرس یک خانه در یک شهر هستند. کامپیوترها با استفاده از آدرس‌های IP مبادرت به یافتن کامپیوترهای دیگر در اینترنت می‌کنند. یک *سرویس‌دهنده سیستم نام دامنه* یا *DNS*[2]، کامپیوتری است که نگهداری پایگاه داده اسامی میزبان‌ها و آدرس‌های IP متناظر با آنها، ترجمه اسامی میزبان‌ها به آدرس‌های IP، را بر عهده دارد. به این عمل ترجمه *DNS lookup* گفته می‌شود. برای مثال، برای دسترسی به وب سایت Deitel، نام میزبان (www.deitel.com) را در مرورگر وب تایپ می‌کنیم. سپس سرویس‌دهنده DNS مبادرت به ترجمه www.deitel.com به آدرس IP سرویس‌دهنده وب Deitel می‌کند (63.110.43.82). آدرس IP، localhost برابر 127.0.01 است.

۵-۱۹ سرویس‌دهنده Apache HTTP

سرویس‌دهنده Apache HTTP توسط Apache Software Foundation پشتیبانی می‌شود و در حال حاضر یکی از محبوبترین سرویس دهنده‌های وب است چرا که از پایداری، هزینه، کارایی و قابلیت حمل مناسبی برخوردار است. این نرم‌افزار یک نرم‌افزار open-source است (به این معنی که کد منبع آن مجاناً و بدون محدودیت در اختیار همه قرار دارد) که بر روی پلاتفرم‌های UNIX، Linux و Windows اجرا می‌شود.

برای برداشت کردن سرویس‌دهنده Apache HTTP از سایت httpd.appache.org بازدید کنید. برای کسب دستورالعمل‌های نصب Apache می‌توانید به وب سایت www.deitel.com یا httpd.apache.org مراجعه کنید. اگر سرویس‌دهنده Apache HTTP بعنوان یک سرویس نصب شده باشد، آماده اجرا پس از نصب است. در غیر اینصورت، می‌توانید سرویس‌دهنده را با انتخاب منوی Start، سپس All Programs > Apache HTTP Server 2.0.52 > Control Apache Server > Start فعال کنید. برای متوقف کردن

1- Top-Level Domain
2- Domain Name System

سرویس‌دهنده Apache مراحل فوق را انجام داده و در پایان گزینه Stop را انتخاب نمائید. برای کاربران Linux، دستورالعمل‌های stop/start سرویس‌دهنده Apache HTTP و اجرای مثال‌ها در وب سایت www.deital.com آورده شده است.

۱۹-۶ تقاضای مستندات XHTML

این بخش شما را با نحوه تقاضای یک مستند XHTML از سرویس‌دهنده Apache HTTP آشنا می‌کند. در ساختار شاخه سرویس‌دهنده Apache HTTP، بایستی مستندات XHTML در شاخه htdocs ذخیره شوند. بر روی سیستم عامل ویندوز، شاخه htdocs در مسیر C:\ProgramsFiles\Apache Group\Apache2 قرار دارد. در سیستم عامل لینوکس، شاخه htdocs در شاخه /user/local/httpd مقیم است. مستند (فایل) test.html را از شاخه مثال‌های فصل ۱۹ که بر روی CD کتاب قرار دارد، کپی کرده و در شاخه htdocs قرار دهید. برای تقاضای مستند، مرورگر وب (همانند Internet Explorer یا Netscape) را اجرا کرده و URL را در فیلد Address وارد سازید (یعنی http://localhost/test.html). شکل ۳-۱۹ نتیجه تقاضای test.html را نشان می‌دهد. [نکته: در Apache، شاخه ریشه سرویس‌دهنده اشاره به شاخه پیش‌فرض، htdocs، دارد از اینرو نام شاخه را قبل از نام فایل (یعنی test.html) در فیلد Address وارد نکرده‌ایم.]

شکل ۳-۱۹ | تقاضای test.html از Apache.

۱۹-۷ معرفی CGI

واسط CGI یا Common Gateway Interface یک پروتکل استاندارد است که به برنامه‌ها امکان می‌دهد تا با سرویس‌دهنده‌های وب و (بطور غیرمستقیم) با سرویس‌گیرنده‌ها (مثلاً مرورگرهای وب) در تعامل قرار گیرند. معمولاً به این برنامه‌ها، برنامه‌های CGI یا اسکریپت‌های CGI می‌گویند. غالباً از CGI برای تولید محتویات دینامیکی وب با استفاده از ورودی سرویس‌گیرنده، پایگاه داده‌ها و سایر سرویس‌های اطلاعاتی، استفاده می‌شود. یک صفحه وب در صورتی دینامیکی است که محتویات آن بصورت برنامه‌نویسی شده در زمان تقاضای برنامه تولید شود، برخلاف محتویات استاتیکی وب که محتویات آن بصورت برنامه‌نویسی شده در زمان تقاضا تولید نمی‌شود.

برای مثل، می‌توانیم از یک صفحه وب استاتیکی استفاده کرده و از کاربر بخواهیم کد پستی را وارد کند، سپس کاربر را به طرف یک اسکریپت CGI هدایت کنیم که برحسب ورودی کاربر یک صفحه وب دینامیک تولید نماید. در این فصل، به معرفی اصول CGI و استفاده از C++ در نوشتن اسکریپت‌های CGI خواهیم پرداخت. CGI اختصاص به یک سیستم عامل خاص و ویژه یا یک زبان برنامه‌نویسی ندارد. CGI به نحوی طراحی شده که می‌تواند با هر زبان برنامه‌نویسی همانند C، C++، Perl، Python یا VisualBasic بکار گرفته شود.

CGI در سال ۱۹۹۳ توسط NCSA (National Centr for Supercomputing Applications) به منظور استفاده با سرویس‌دهنده وب HTTPd توسعه یافته است. برخلاف پروتکل‌های وب و زبان‌های که دارای ساختار رسمی هستند، توصیف اولیه CGI که توسط NCSA نوشته شده اثبات کرده است که CGI بعنوان یک استاندارد غیررسمی در جهان پذیرفته شده است. پشتیبانی از CGI بسرعت در میان سرویس‌دهنده‌های وب از جمله Apache پذیرفته شده است.

۱۹-۸ تراکنش ساده HTTP

قبل از بررسی نحوه عملکرد CGI، درک اولیه‌ای از شبکه و وب گسترده جهانی (www[1]) ضروری است. در این بخش، به بررسی عملکرد داخلی *پروتکل انتقال فوق متن* (HTTP[2]) و اتفاقاتی که در پس پرده نمایش یک صفحه وب در مرورگر رخ می‌دهد، می‌پردازیم. HTTP مشخص کننده مجموعه‌ای از *متدها* و *سرآیندها* (*headers*) است که به سرویس‌گیرنده‌ها و سرویس‌دهنده‌ها امکان می‌دهند تا با یکدیگر به تعامل پرداخته و اطلاعات را به یک شکل واحد و به روش قابل اطمینانی مبادله نمایند.

در ساده‌ترین فرم، یک صفحه وب چیزی بیش از یک مستند HTML نیست. این مستند یک فایل متنی ساده حاوی *نشانه‌ها* یا *دنباله‌ها* است که به مرورگر وب نشان می‌دهند چگونه اطلاعات موجود در مستند را به نمایش در آورد. برای مثال، به نشانه HTML زیر توجه کنید:

```
<title> My Web Page </title>
```

این نشانه، به مرورگر اعلان می‌کند که عبارت قرار گرفته مابین دو دنباله شروع و پایانی (**<title>** و **</title>**) عنوان صفحه وب است. همچنین مستندات HTML می‌توانند حاوی داده‌های *فوق متن* (با نام *فوق لینک* نیز شناخته می‌شوند) باشند، که لینک‌هایی به صفحات مختلف یا بخش‌های دیگری در همان صفحه ایجاد می‌کنند. هنگامی که کاربر بر روی یک فوق لینک کلیک می‌کند، صفحهٔ درخواستی وب (یا بخش متفاوتی از همان صفحه وب) به بدورن پنجره مرورگر کاربر بار می‌شود.

هر مستندی که برای نمایش بر روی وب در دسترس می‌باشد، دارای یک URL[3] است که در واقع آدرسی است که موقعیت یا مکان منبع را نشان می‌دهد. URL حاوی اطلاعاتی است که مرورگر را به منبع مستند که کاربر مایل به دسترسی به آن است، هدایت می‌کند. کامپیوترهایی که نرم‌افزار سرویس‌دهنده وب بر روی آنها اجرا می‌شود چنین منابعی را فراهم می‌آورند. اجازه دهید تا به بررسی اجزای URL زیر بپردازیم:

```
http://www.deitel.com/books/downlods.htm
```

1- World Wide Web
2- HyperText Transfer Protocol
3- Uniform Resource Locator

http:// نشان می‌دهد که منبع از پروتکل HTTP استفاده می‌کند. بخش میانی، **www.deitel.com** نام کامل میزبان سرویس‌دهنده است. *نام میزبان*، نام کامپیوتری است که منابع بر روی آن قرار دارند. معمولاً به این کامپیوترها، *میزبان* گفته می‌شود، چرا که منزلگاه و نگهدارنده منابع هستند. نام میزبان **www.deitel.com** به یک *آدرس IP* ترجمه می‌شود (**63.110.43.82**)، که نشاندهنده هویت سرویس‌دهنده می‌باشد، این هویت همانند یک شماره تلفن است که منحصراً متعلق به یک خط تلفن می‌باشد. معمولاً ترجمه نام میزبان به یک آدرس IP توسط یک *سرویس‌دهنده نام دامنه* (DNS[1]) صورت می‌گیرد. DNS به کامپیوتری اطلاق می‌شود که پایگاه داده‌ای (بانک اطلاعاتی) از اسامی میزبان‌ها و آدرس‌های IP آنها را در خود نگهداری می‌کند. به این فرآیند ترجمه *DNS lookup* گفته می‌شود.

مابقی URL نشاندهنده *نام منبع* درخواستی یعنی **books/downloads.htm/** (یک مستند HTML) است. این بخش از URL هم نام منبع (**downloads.htm**) و هم مسیر یا مکان آنرا (**books/**) بر روی سرویس‌دهنده وب مشخص کرده است. مسیر بکار رفته می‌تواند نشاندهنده یک شاخه واقعی بر روی سیستم فایل سرویس‌دهنده وب باشد. با این وجود، به دلایل امنیتی، غالباً مسیر مشخص شده، نشاندهنده یک شاخه مجازی است. در چنین سیستم‌هایی، سرویس‌دهنده مبادرت به تبدیل شاخه مجازی به یک مکان واقعی بر روی سرویس‌دهنده (یا کامپیوتر دیگری بر روی شبکه سرویس‌دهنده) می‌کند، از اینرو مکان واقعی منابع پنهان خواهد ماند. علاوه بر این، برخی از منابع بصورت دینامیکی ایجاد می‌شوند و در هیچ کجای کامپیوتر سرویس‌دهنده قرار ندارند.

به هنگام وارد کردن یک URL، مرورگر مبادرت به انجام یک تراکنش ساده HTTP برای دریافت و نمایش صفحه وب مورد تقاضا می‌کند. در شکل ۴-۱۹ جزئیات تراکنش صورت گرفته، دیده می‌شود. این تراکنش متشکل از تأثیر متقابل مابین مرورگر وب (طرف سرویس‌گیرنده یا *Client*) و برنامه کاربردی سرویس‌دهنده وب (طرف سرویس‌دهنده یا *server*) می‌باشد. در شکل ۴-۱۹، مرورگر وب یک تقاضای HTTP به سرویس‌دهنده ارسال کرده است. تقاضا (در ساده‌ترین فرم) عبارت است از

```
GET /books/downloads.htm HTTP/1.1
Host: wwww.deitel.com
```

کلمه ***GET*** یک *متد HTTP* است که نشان می‌دهد سرویس‌گیرنده مایل به بدست آوردن منبعی از سرویس‌دهنده است. مابقی تقاضا، تدارک بیننده نام مسیر منبع (یک مستند HTML) و نام پروتکل و شماره نسخه آن می‌باشد (**HTTP/1.1**).

1- Domain Name Server

هر سرویس‌دهنده‌ای که قادر به درک HTTP (نسخه 1.1) باشد می‌تواند این تقاضا را بررسی کرده و بطور مناسب به آن پاسخ دهد. در شکل ۴-۱۹ نتیجه درخواست دیده می‌شود. ابتدا سرویس‌دهنده با ارسال یک عبارت متنی که نشاندهنده نسخه HTTP و بدنبال آن یک کد عددی و کلمه تعیین وضعیت تراکنش است به سرویس‌گیرنده پاسخ می‌دهد. برای مثال

```
HTTP/1.1 200 OK
```

نشاندهنده موفقیت آمیز بودن عمل است در حالیکه

```
HTTP/1.1 404 Not found
```

به سرویس‌گیرنده اعلان می‌کند که سرویس‌دهنده وب نتوانسته است منبع درخواستی را پیدا کند.

شکل ٤-١٩ | تعامل سرویس‌گیرنده با سرویس‌دهنده. گام 1: تقاضای GET،
`GET/books/downloads.htm/HTTP/1.1`

شکل ٤-١٩ | تعامل سرویس‌گیرنده با سرویس‌دهنده. گام 2: پاسخ HTTP، HTTP/1.1 200 OK.

سپس سرویس‌دهنده یک یا چند سرآیند HTTP ارسال می‌کند که حاوی اطلاعات بیشتری در مورد داده‌ای است که ارسال خواهد شد. در این مورد، سرویس‌دهنده یک مستند متنی HTML ارسال می‌کند، از اینرو سرآیند HTTP برای این مثال بصورت زیر خواهد بود:

```
Content-type: text/html
```

اطلاعات تدارک دیده شده در این سرآیند مشخص کننده نوع محتویات *MIME*[1] است که سرویس‌دهنده آنها را به مرورگر انتقال می‌دهد. MIME یکی از استانداردهای اینترنت است که روش قالبندی داده‌های مشخصی را تعیین می‌کند تا برنامه‌ها بتوانند داده‌ها را بطرز صحیحی تفسیر نمایند. برای مثال، **text/plain** از نوع MIME است که نشان می‌دهد اطلاعات ارسالی متنی است که می‌تواند مستقیماً و بدون هیچ گونه تفسیری به نمایش درآید. به همین ترتیب نوع **image/gif** نشان می‌دهد که محتوی یک تصویر GIF است. زمانیکه مرورگر این نوع از MIME را دریافت کند، مبادرت به نمایش تصویر خواهد کرد.

1- Multipurpose Internet Mail Extensions

یک خط خالی که پس از سرآیند یا سرآیندها قرار می‌گیرد به سرویس‌گیرنده اعلان می‌کند که سرویس‌دهنده به ارسال سرآیندهای HTTP خاتمه داده است. سپس سرویس‌دهنده محتویات مستند HTML تقاضا شده (**downloads.htm**) را ارسال می‌کند. پس از کامل شدن انتقال منبع، سرویس‌دهنده به اتصال برقرار شده، خاتمه می‌دهد. در این نقطه، مرورگر طرف سرویس‌گیرنده شروع به تجزیه HTML دریافتی و راندو (یا نمایش) آن می‌کند.

۹-۱۹ اسکریپت‌های ساده CGI

مادامیکه یک فایل XHTML بر روی سرویس‌دهنده بدون تغییر باقی بماند، URL مرتبط با آن، همان محتویات را در هر بار دسترسی فایل در مرورگر سرویس‌گیرنده به نمایش در خواهد آورد. برای اعمال تغییر در محتویات یک فایل XHTML (مثلاً افزودن لینک‌های جدید)، بایستی فردی فایل را بصورت دستی بر روی سرویس‌دهنده و احتمالاً با استفاده از یک برنامه ویرایشگر متن یا نرم‌افزار طراحی صفحه وب تغییر دهد. این نحوه تغییر برای مولفان صفحه وب که می‌خواهند صفحه وب دینامیکی جالبی ایجاد کند، مشکل است. داشتن فردی که مدام صفحه وب را تغییر دهد، کاری خسته کننده است. برای مثال، اگر بخواهید صفحه وب شما همیشه تاریخ جاری یا شرایط آب و هوا را به نمایش درآورد، باید این صفحه مرتباً به روز شود.

اولین اسکریپت CGI

اولین اسکریپت CGI ما در شکل ۵-۱۹ آورده شده است. توجه کنید که برنامه بیشتر از عبارات **cout** تشکیل شده است (خطوط 26-19). تا بدین جا، همیشه خروجی **cout** بر روی صفحه نمایش ظاهر می‌شد. با این وجود، به لحاظ تکنیکی، مکان یا هدف پیش‌فرض برای **cout**، خروجی استاندارد است. زمانیکه یک برنامه C++ بعنوان یک اسکریپت CGI اجرا می‌شود، خروجی استاندارد، توسط سرویس‌دهنده وب به سمت مرورگر وب سرویس‌گیرنده هدایت می‌شود. برای اجرای برنامه بعنوان یک اسکریپت CGI، فایل اجرائی کامپایل شده C++ را در شاخه **cgi-bin** سرویس‌دهنده وب قرار داده‌ایم. برای برآورده کردن اهداف این فصل، پسوند فایل اجرای را از **exe.** با **cgi.** تغییر داده‌ایم. با فرض اینکه سرویس‌دهنده وب بر روی کامپیوتر محلی شما قرار دارد، می‌توانید اسکریپت را با تایپ

```
http://localhost/cgi-bin/localtime.cgi
```

در فیلد Address یا Location مرورگر خود، به اجرا در آورید. اگر این اسکریپت را از یک سرویس‌دهنده وب راه دور تقاضا کنید، نیاز دارید تا localhost را با نام ماشین سرویس‌دهنده یا آدرس IP آن جایگزین سازید.

```
1  // Fig. 19.5: localtime.cpp
2  // Displays the current date and time in a Web browser.
3  #include <iostream>
4  using std::cout;
```

```
5
6  #include <ctime>//definitions of time_t, time, localtime and asctime
7  using std::time_t;
8  using std::time;
9  using std::localtime;
10 using std::asctime;
11
12 int main()
13 {
14    time_t currentTime; // variable for storing time
15
16    cout << "Content-Type: text/html\n\n"; // output HTTP header
17
18    // output XML declaration and DOCTYPE
19    cout << "<?xml version = \"1.0\"?>"
20       << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.1//EN\" "
21       << "\"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd\">";
22
23    time( &currentTime ); // store time in currentTime
24
25    // output html element and some of its contents
26    cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\">"
27       << "<head><title>Current date and time</title></head>"
28       << "<body><p>" << asctime( localtime( &currentTime ) )
29       << "</p></body></html>";
30    return 0;
31 } // end main
```

شکل ۵-۱۹ | اولین اسکریپت CGI.

نظریه خروجی استاندارد شبیه به ورودی استاندارد است، که آنرا مرتبط با **cin** می‌دانیم. همانطور که ورودی استاندارد اشاره به منبع استاندارد ورودی به برنامه (معمولاً صفحه‌کلید) دارد، خروجی استاندارد اشاره به مقصد استاندارد خروجی از برنامه (معمولاً صفحه نمایش) دارد. امکان هدایت (یا لوله‌کشی) خروجی استاندارد به مقصد دیگر وجود دارد. از اینرو، اسکریپت CGI ما، زمانیکه یک سرآیند HTTP (خط 16) یا عناصر XHTML (خطوط 21-19 و 29-26) را خارج می‌سازیم، خروجی به سرویس‌دهنده وب ارسال می‌شود. سرویس‌دهنده آن خروجی را به سرویس‌دهنده از طریق HTTP ارسال می‌کند که، سرآیند و عناصر را تفسیر می‌کند.

نوشتن یک برنامه C++ که زمان و تاریخ جاری را چاپ کند کار سختی نیست. در واقع اینکار مستلزم نوشتن چند خط کد است (خطوط 23,14 و 28). خط 14 مبادرت به اعلان متغیر **currentTime** بعنوان متغیری از نوع **time_t** کرده است. تابع **time** در خط 23 زمان جاری را بدست آورده و مقدار آنرا در مکان مشخص شده توسط پارامتر ذخیره می‌کند (در این مورد **currentTime**). تابع کتابخانه‌ای **localtime** (خط 28)، زمانیکه متغیر **time_t** را می‌پذیرد (یعنی **currentTime**) یک اشاره‌گر به شی که حاوی زمان محلی است برگشت می‌دهد. تابع **asctime** در خط 28 که یک اشاره‌گر به شی که حاوی زمان است دریافت می‌کند، رشته‌ای بصورت زیر برگشت می‌دهد

```
Wed Oct 31 13:10:37 2007
```

اگر بخواهیم زمان جاری را به پنجره مرورگر یک سرویس‌گیرنده ارسال کنیم چه باید کرد؟ CGI این امکان را با هدایت خروجی یک برنامه به خود سرویس‌دهنده وب فراهم می‌آورد که آن خروجی هم به مرورگر سرویس‌گیرنده ارسال می‌شود.

نحوه هدایت خروجی توسط سرویس‌دهنده وب

در شکل ۶-۱۹ این فرآیند بدقت بررسی شده است. در گام اول، سرویس‌گیرنده تقاضای منبعی بنام **localtime.cgi** از سرویس‌دهنده می‌کند، همانند تقاضای صورت گرفته برای **downloads.html** در مثال قبلی (شکل ۴-۱۹). اگر سرویس‌دهنده برای کار با اسکریپت CGI پیکربندی نشده باشد، می‌تواند محتویات فایل اجرای C++ را به سرویس‌دهنده برگشت دهد، مثل اینکه مستند دیگری وجود ندارد. با این همه، براساس پیکربندی سرویس‌دهنده وب، سرویس‌دهنده **localtime.cgi** (پیاده‌سازی شده توسط C++) را اجرا کرده و خروجی اسکریپت CGI را به مرورگر وب ارسال می‌کند.

به هر حال اگر سرویس‌دهنده وب بدرستی پیکربندی شده باشد، انواع منابع مختلف را که بایستی به روش‌های متفاوتی با آنها کار شود را تشخیص خواهد داد. برای مثال، زمانیکه منبع یک اسکریپت CGI باشد، بایستی اسکریپت توسط سرویس‌دهنده قبل از ارسال آن، اجرا شده باشد. یک اسکریپت CGI به یکی از دو روش معین می‌شوند: خواه دارای یک پسوند نام فایل ویژه است (همانند **cgi.** یا **exe.**) یا در شاخه خاص قرار دارد (اغلب **cgi-bin**).

علاوه بر این، بایستی مدیر سرویس‌دهنده به صراحت مجوزهای لازم را در اختیار سرویس‌گیرنده‌های راه دور قرار دهد تا آنها بتوانند به اسکریپت‌های CGI دسترسی پیدا کرده و به اجرا در آورند.

در گام دوم از شکل ۶-۱۹، سرویس‌دهنده تشخیص داده است که منبع یک اسکریپت CGI است و اسکریپت را اجرا می‌کند. در گام سوم، خروجی توسط سه عبارت **cout** تولید شده (خطوط 16، 19-21 و 26-29 از شکل ۵-۱۹) و به خروجی استاندارد ارسال شده و به سرویس‌دهنده وب برگشت داده می‌شود. سرانجام در گام چهارم، سرویس‌دهنده وب پیغامی به خروجی اضافه می‌کند که دلالت بر وضعیت تراکنش HTTP دارد (همانند HTTP/1.1 200 Ok برای موفقیت) و کل خروجی را از برنامه CGI به سرویس‌گیرنده ارسال می‌کند.

شکل ٦-١٩ | گام اول: تقاضای GET/cgi-bin/localtime.cgi HTTP/1.1.

شکل ٦-١٩ | گام دوم: سرویس‌دهنده وب اسکریپت CGI را راه‌اندازی می‌کند.

شکل ٦-١٩ | گام سوم: خروجی اسکریپت به سرویس‌دهنده وب ارسال می‌شود.

سپس مرورگر طرف سرویس‌گیرنده مبادرت به پردازش مستند XHTML کرده و نتیجه را به نمایش در می‌آورد. توجه داشته باشید که مرورگر از آنچه که در سرویس‌دهنده اتفاق می‌افتد بی‌خبر است. به عبارتی

دیگر، تا آنجا که به مرورگر مربوط می‌شود، وی تقاضای منبعی را مطرح کرده و پاسخی را دریافت می‌کند. مرورگر خروجی اسکریپت را دریافت و تفسیر می‌کند.

شکل ٦-١٩ | گام چهارم: پاسخ HTTP، HTTP/1.1 200 OK.

در واقع، می‌توانید با اجرای **localtime.cgi** از طریق خط فرمان، شاهد محتوی باشید که مرورگر دریافت می‌کند. شکل ۷-۱۹ نمایشی از خروجی است. به منظور برآورده کردن اهداف این فصل، خروجی را برای اینکه قابل فهم باشد، قالب‌بندی کرده‌ایم. توجه کنید که در اسکریپت CGI، بایستی خروجی شامل سرآیند Content-Type باشد، از آنجاییکه در یک مستند XHTML، سرویس‌دهنده وب این سرآیند را شامل می‌شود.

اسکریپت CGI سرآیند Content-Type، یک خط خالی و داده (XHTML، متن ساده و غیره) را در خروجی استاندارد چاپ می‌کند. زمانیکه اسکریپت CGI بر روی سرویس‌دهنده وب اجرا می‌شود، سرویس‌دهنده خروجی اسکریپت را بازیابی کرده، پاسخ HTTP را به ابتدای آن وارد و محتوی را به سرویس‌گیرنده ارسال می‌کند.

```
Content-Type: text/html

<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
     http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd>

<html xmlns = http://www.w3.org/1999/xhtml>
     <head>
          <title>Current date and time</title>
     </head>

     <body>
          <p>Wet Oct 13 10:22:18 2004</p>
     </body>
</html>
```

شکل ۷-۱۹ | خروجی localtime.cgi به هنگام اجرا از طریق خط فرمان.

نمایش متغیرهای محیطی

برنامه شکل ۸-۱۹ مبادرت به نمایش متغیرهای محیطی می‌کند که سرویس‌دهنده Apache HTTP برای اسکریپت‌های CGI تنظیم می‌نماید. این متغیرها حاوی اطلاعاتی در مورد محیط سرویس‌گیرنده و سرویس‌دهنده، همانند نوع مرورگر وب بکار رفته و مکان مستندی بر روی سرویس‌دهنده هستند. خطوط 14-23 آرایه رشته‌ای با اسامی متغیرهای محیطی CGI را مقداردهی اولیه کرده‌اند. خط 37 آغاز جدول XHTML است که داده‌ها در آن به نمایش در می‌آیند.

```
1  // Fig. 19.8: environment.cpp
2  // Program to display CGI environment variables.
3  #include <iostream>
4  using std::cout;
5
6  #include <string>
```

```
7  using std::string;
8
9  #include <cstdlib>
10 using std::getenv;
11
12 int main()
13 {
14    string environmentVariables[ 24 ] = {
15       "COMSPEC", "DOCUMENT_ROOT", "GATEWAY_INTERFACE",
16       "HTTP_ACCEPT", "HTTP_ACCEPT_ENCODING",
17       "HTTP_ACCEPT_LANGUAGE", "HTTP_CONNECTION",
18       "HTTP_HOST", "HTTP_USER_AGENT", "PATH",
19       "QUERY_STRING", "REMOTE_ADDR", "REMOTE_PORT",
20       "REQUEST_METHOD", "REQUEST_URI", "SCRIPT_FILENAME",
21       "SCRIPT_NAME", "SERVER_ADDR", "SERVER_ADMIN",
22       "SERVER_NAME","SERVER_PORT","SERVER_PROTOCOL",
23       "SERVER_SIGNATURE","SERVER_SOFTWARE" };
24
25    cout << "Content-Type: text/html\n\n"; // output HTTP header
26
27    // output XML declaration and DOCTYPE
28    cout << "<?xml version = \"1.0\"?>"
29       << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.1//EN\" "
30       << "\"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd\">";
31
32    // output html element and some of its contents
33    cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\">"
34       << "<head><title>Environment Variables</title></head><body>";
35
36    // begin outputting table
37    cout << "<table border = \"0\" cellspacing = \"2\">";
38
39    // iterate through environment variables
40    for ( int i = 0; i < 24; i++ )
41    {
42       cout << "<tr><td>" << environmentVariables[i] << "</td><td>";
43
44       // attempt to retrieve value of current environment variable
45       char *value = getenv( environmentVariables[ i ].c_str() );
46
47       if ( value != 0 ) // environment variable exists
48          cout << value;
49       else
50          cout << "Environment variable does not exist.";
51
52       cout << "</td></tr>";
53    } // end for
54
55    cout << "</table></body></html>";
56    return 0;
57 } // end main
```

شکل ۸-۱۹ | بازیابی متغیرهای محیطی از طریق تابع getenv.

خطوط 52-42 هر سطر از جدول را چاپ می‌کنند. اجازه دهید تا از نزدیک به بررسی این خطوط بپردازیم. خط 42 یک دنباله آغازین <**tr**> (سطر جدول) را که نشاندهنده ابتدای یک سطر جدید در جدول است را در خروجی قرار می‌دهد. خط 52 دنباله پایانی متناظر </**tr**> را که دلالت بر انتهای سطر دارد، در خروجی قرار می‌دهد. هر سطر از جدول حاوی دو سلول است که برای نام متغیر محیطی و داده

مرتبط با آن متغیر در نظر گرفته شده‌اند. دنباله شروع **<td>** در خط 42 آغاز یک سلول جدید در جدول است. حلقه **for** در خطوط 53-40، در میان 24 شی رشته حرکت می‌کند. نام هر متغیر محیطی در سمت چپ سلول به نمایش در می‌آید (خط 42). خط 45 مبادرت به بازیابی مقدار مرتبط با متغیر محیطی توسط فراخوانی تابع **getenv** از **<cstdlib>** کرده و مقدار رشته‌ای برگشت داده شده از فراخوانی تابع **environmentVariables[i].c_str()** است. تابع **getenv** یک رشته ***** **char** حاوی مقدار یک متغیر محیطی مشخص شده را برگشت می‌دهد یا اگر متغیر محیطی وجود نداشته باشد، اشاره‌گر **null** برگشت می‌دهد.

خطوط 50-47 محتویات را در سلول راست چاپ می‌کنند. اگر متغیر محیطی وجود داشته باشد (یعنی **getenv** اشاره‌گر **null** برگشت ندهد)، خط 48 مقدار برگشتی توسط تابع **getenv** را چاپ می‌کند. اگر متغیر محیطی وجود نداشته باشد، خط 50 پیغام مناسبی چاپ می‌نماید. اجرای نمونه این برنامه در شکل ۸-۱۹ به هنگام اجرای این مثال بر روی سرویس‌دهنده Apache HTTP آورده شده است.

۱۰-۱۹ ارسال ورودی به اسکریپت CGI

اگرچه متغیرهای محیطی از پیش تنظیم شده اطلاعات زیادی دارند، اما می‌خواهیم انواع مختلفی از اطلاعات را در اسکریپت CGI خود همانند نام کاربر یا پرس‌وجوی موتور جستجو داشته باشیم. متغیر محیطی **QUERY_STRING** مکانیزمی است که اینکار را انجام می‌دهد. متغیر **QUERY_STRING** حاوی اطلاعاتی است که به URL در ضمن یک تقاضا الصاق می‌شود. برای مثال، URL

```
www.somesite.com/cgi-bin/script.cgi?state=California
```

سبب می‌شود تا مرورگر وب تقاضای یک اسکریپت (cgi-bin/script.cgi) را با رشته پرس‌وجو (state=California) از www.somesite.com انجام دهد. سرویس‌دهنده وب رشته پرس‌وجوی پس از **?** را در متغیر محیطی **QUERY_STRING** ذخیره می‌سازد. رشته پرس‌وجو پارامترهای فراهم می‌آورد که تقاضا برای یک سرویس‌گیرنده مشخص را بهینه‌سازی می‌کنند. دقت کنید که علامت سوال (**?**) بخشی از منبع درخواستی و رشته پرس‌وجو نمی‌باشد. این کاراکتر فقط نقش جدا کننده مابین این دو را بازی می‌کند.

برنامه شکل ۹-۱۹ مثال ساده‌ای از یک اسکریپت CGI است که داده را خوانده و از طریق **QUERY_STRING** ارسال می‌کند. به روش‌های مختلف می‌توان رشته پرس‌وجو را قالب‌بندی کرد. اسکریپت CGI که رشته پرس‌وجو را می‌خواند باید از نحوه تفسیر داده قالب‌بندی شده مطلع باشد. در مثال شکل ۹-۱۹ رشته پرس‌وجو حاوی دنباله‌ای از جفت‌های مقدار–نام است که توسط آمپرسنج (**&**) بصورت **name=Jill&age=22** از هم متمایز شده‌اند.

در خط 16 از شکل ۹-۱۹ رشته "QUERY_STRING" به تابع **getenv** ارسال شده که رشته پرس‌وجو یا اشاره‌گر **null** را در صورتیکه سرویس‌دهنده، متغیر محیطی **QUERY_STRING** را تنظیم نکرده باشد، برگشت می‌دهد. اگر این متغیر محیطی وجود داشته باشد (یعنی **getenv** اشاره‌گر **null** برگشت ندهد)، خط 17 مجزا **getenv** را فراخوانی می‌کند. این بار رشته پرس‌وجوی برگشتی به متغیر رشته‌ای **query** تخصیص داده می‌شود. پس از قرار دادن سرآیند و برخی از دنباله‌های شروع XHTML و عنوان (خطوط 19-29) به بررسی اینکه در **query** داده وجود دارد یا خیر می‌پردازیم (خط 32). اگر چنین نباشد، پیغامی به کاربر عرضه می‌شود و از وی می‌خواهد تا یک رشته پرس‌وجو به URL اضافه کند. همچنین یک لینک به URL افزوده‌ایم که شامل یک رشته پرس‌وجوی ساده است. داده رشته پرس‌وجو می‌تواند تعیین کننده یک فوق لینک در صفحه وب در زمان کدگشایی باشد. محتویات رشته پرس‌وجو توسط خط 36 چاپ می‌شود.

این مثال ساده به توصیف نحوه دسترسی داده ارسالی به اسکریپت CGI در رشته پرس‌وجو پرداخته است. در ادامه این فصل با مثال‌های آشنا خواهید شد که نحوه تقسیم یک رشته پرس‌وجو به قسمتهای سودمندتر اطلاعاتی را نشان می‌دهند که می‌توان با استفاده از متغیرهای مجزا از آنها نگهداری کرد.

۱۹-۱۱ استفاده از فرم‌های XHTML برای ارسال ورودی

داشتن سرویس‌گیرنده‌های که مستقیماً ورودی را وارد URL می‌کند روش چندان کاربرپسندی نیست. خوشبختانه، XHTML قابلیتی را از طریق فرم‌ها در صفحات وب فراهم آورده که می‌تواند روش بسیار مناسبی برای کاربران در وارد کردن اطلاعاتی باشند که به یک اسکریپت CGI ارسال خواهند شد.

```
1  // Fig. 19.9: querystring.cpp
2  // Demonstrating QUERY_STRING.
3  #include <iostream>
4  using std::cout;
5
6  #include <string>
7  using std::string;
8
9  #include <cstdlib>
10 using::getenv;
11
12 int main()
13 {
14    string query = "";
15
16    if ( getenv( "QUERY_STRING" ) ) // QUERY_STRING variable exists
17       query = getenv( "QUERY_STRING" );//retrieve QUERY_STRING value
18
19    cout << "Content-Type: text/html\n\n"; // output http header
20
21    // output XML declaration and DOCTYPE
22    cout << "<?xml version = \"1.0\"?>"
23       << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.1//EN\" "
24       << "\"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd\">";
25
```

```
26    // output html element and some of its contents
27    cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\">"
28       << "<head><title>Name/Value Pairs</title></head><body>";
29    cout << "<h2>Name/Value Pairs</h2>";
30
31    // if query contained no data
32    if ( query == "" )
33       cout<<"Please add some name-value pairs to the URL above.<br/> Or"
34         <<" try <a href=\"querystring.cgi?name=Joe&age=29\">this</a>.";
35    else // user entered query string
36       cout << "<p>The query string is: " << query << "</p>";
37
38    cout << "</body></html>";
39    return 0;
40 } // end main
```

شکل ۹-۱۹ | خواندن ورودی از QUERY_STRING.

عنصر form

عنصر **form** یک فرم XHTML بوجود می‌آورد. معمولاً این عنصر دو صفت دریافت می‌کند. صفت اول **action** است که مشخص کننده منبع سرویس‌دهنده برای اجرا شدن در زمانی است که کاربر فرم را تسلیم می‌کند. برای اهداف ما، معمولاً **action** یک اسکریپت CGI خواهد بود که داده فرم را پردازش می‌نماید. صفت دوم که در عنصر **form** بکار گرفته می‌شود، **method** است، که شناسه تقاضای HTTP می‌باشد (یعنی **get** یا **post**) تا به هنگام تسلیم فرم توسط مرورگر به سرویس‌دهنده وب بکار گرفته شود. در این بخش مثال‌های با استفاده از هر دو نوع تقاضای **get** و **post** مطرح کرده‌ایم. یک فرم XHTML می‌تواند حاوی هر تعداد از عناصر باشد. جدول شکل ۱۰-۱۹ بطور خلاصه به معرفی چند عنصر فرم پرداخته است.

نام عنصر	نوع صفت	توضیح
input	**text**	یک فیلد تک خطی برای وارد کردن متن فراهم می‌آورد.
	password	همانند **text** است، اما هر کاراکتر تایپ شده را بصورت ستاره (*) ظاهر می‌کند.
	checkbox	یک جعبه‌چک به نمایش در می‌آورد که می‌تواند انتخاب شود **(true)** یا از انتخاب خارج گردد **(false)**.
	radio	دکمه‌های رادیوی همانند جعبه‌چک‌ها هستند بجز اینکه فقط یک دکمه رادیویی در گروه دکمه‌های رادیویی می‌تواند در هر بار انتخاب شود.
	button	یک دکمه به نمایش در می‌آورد.
	submit	یک دکمه است که داده فرم را مطابق **action** فرم تسلیم می‌کند.
	image	همانند **submit** است، اما بجای دکمه یک تصویر به نمایش در می‌آورد.
	reset	یک دکمه به نمایش در می‌آورد که فیلدهای فرم را به مقادیر پیش‌فرض آنها باز می‌گرداند.
	file	یک فیلد متنی و دکمه به نمایش در می‌آورد که به کاربر امکان می‌دهد تا فایلی

	را به سرویس‌دهنده وب ارسال کند **(upload)**. زمانیکه کلیک شود، یک کادر تبادلی فایل باز می‌شود و به کاربر امکان می‌دهد تا فایل را انتخاب کند.
Hidden	داده فرمی را که می‌تواند توسط رسیدگی کننده فرم در سرویس‌دهنده بکار گرفته شود، پنهان می‌سازد. این ورودی‌ها در دید کاربر قرار ندارند.
select	یک منوی پایین افتادنی یا جعبه انتخاب به نمایش در می‌آورد.
textarea	فیلد متنی مضاعف بدست می‌دهد. متن می‌تواند در آن وارد یا به نمایش در آید.

شکل ۱۰-۱۹ | عناصر فرم در XHTML

تقاضای get

برنامه شکل ۱۱-۱۹ به بررسی یک فرم XHTML با استفاده از روش **HTTP get** پرداخته است. فرم توسط خطوط 36-34 با عنصر form بوجود می‌آید. توجه کنید که صفت **method** دارای مقدار "get" و صفت **action** دارای مقدار "getquery.cgi" است (در واقع اسکریپت خود را برای رسیدگی به داده فرم پس از تسلیم، فراخوانی خواهد کرد).

فرم حاوی دو فیلد **input** است. ورودی اول (خط 35) یک فیلد متنی تک خطی بنام word است (**type="text"**). ورودی دوم (خط 36) دکمه‌ای را با برچسب Submit Word به نمایش درمی‌آورد که داده‌ای فرم را تسلیم می‌کند (**value="Submit Word"**).

اولین بار که اسکریپت اجرا می‌شود. هیچ مقداری در **QUERY_STRING** وجود ندارد. (مگر آنکه کاربر رشته پرس‌وجو را به URL الحاق کرده باشد)

```
1  // Fig. 19.11: getquery.cpp
2  // Demonstrates GET method with XHTML form.
3  #include <iostream>
4  using std::cout;
5
6  #include <string>
7  using std::string;
8
9  #include <cstdlib>
10 using std::getenv;
11
12 int main()
13 {
14    string nameString = "";
15    string wordString = "";
16    string query = "";
17
18    if (getenv("QUERY_STRING"))//QUERY_STRING variable exists
19       query = getenv( "QUERY_STRING" );// retrieve QUERY_STRING value
20
21    cout << "Content-Type: text/html\n\n"; // output HTTP header
22
23    // output XML declaration and DOCTYPE
24    cout << "<?xml version = \"1.0\"?>"
25       << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.1//EN\" "
26       << "\"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd\">";
```

```
27
28    // output html element and some of its contents
29    cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\">"
30       << "<head><title>Using GET with Forms</title></head><body>";
31
32    // output xhtml form
33    cout << "<p>Enter one of your favorite words here:</p>"
34       << "<form method = \"get\" action = \"getquery.cgi\">"
35       << "<input type = \"text\" name = \"word\"/>"
36       << "<input type = \"submit\" value = \"Submit Word\"/></form>";
37
38    if ( query == "" ) // query is empty
39       cout << "<p>Please enter a word.</p>";
40    else // user entered query string
41    {
42       int wordLocation = query.find_first_of( "word=" ) + 5;
43       wordString = query.substr( wordLocation );
44
45       if ( wordString == "" ) // no word was entered
46          cout << "<p>Please enter a word.</p>";
47       else // word was entered
48          cout << "<p>Your word is: " << wordString << "</p>";
49    } // end else
50
51    cout << "</body></html>";
52    return 0;
53 } // end main
```

شکل ۱۱-۱۹ | استفاده از روش get به همراه فرم XHTML.

زمانیکه کاربر عبارتی را در فیلد متنی word وارد و بر روی دکمه Submit Word کلیک کرد، اسکریپت مجدداً تقاضا می‌شود. این بار، نام فی‌لد ورودی (word) و مقدار وارد شده توسط کاربر در متغیر محیطی **QUERY_STRING** جای خواهند داشت. یعنی اگر کاربر کلمه "technology" را وارد و بر روی Submit Word کلیک کند به **QUERY_STRING** مقدار **value=technology** تخصیص می‌یابد. دقت کنید که رشته پرس‌وجو به URL در فیلد Address مرورگر به همراه علامت سوال (**?**) قبل از آن افزوده می‌شود.

در اجرای دوم اسکریپت، رشته پرس‌وجو کدگشایی می‌شود. خط 42 از متد **find_first_of** برای جستجوی پرس‌وجو به منظور یافتن اولین پیشامد از **word=** استفاده کرده است، که یک مقدار صحیح برگشت می‌دهد که نشاندهنده موقعیت آن در رشته است. سپس خط 42 مقدار 5 را به مقدار برگشتی توسط **find_first_of** اضافه می‌کند تا **wordLocation** با موقعیت اولین کاراکتر وارد شده توسط کاربر تنظیم شود. تابع **substr** در خط 43 مابقی رشته آغاز شونده از **wordLocation** را برگشت می‌دهد. خط 45 تعیین می‌کند که کاربر کلمه‌ای وارد کرده است یا خیر. اگر چنین باشد، خط 48 کلمه وارد شده توسط کاربر را چاپ می‌کند.

تقاضای post

در دو مثال قبلی از روش **get** برای ارسال داده به اسکریپت‌های CGI از یک متغیر محیطی استفاده کردیم. عموماً مرورگرهای وب با سرویس‌دهنده‌های وب توسط فرم‌های تسلیم شده به روش **HTTP post** در تعامل قرار می‌گیرند. برنامه‌های CGI محتویات تقاضاهای **post** را با استفاده از استاندارد ورودی می‌خوانند. برای اینکه مقایسه‌ای انجام دهیم، اجازه دهید مجدداً برنامه ۱۱-۱۹ را با استفاده از روش **post** پیاده‌سازی کنیم (بعنوان برنامه شکل ۱۲-۱۹). توجه کنید کد بکار رفته در دو برنامه واقعاً یکسان هستند فرم XHTML در خطوط 45-43 نشان می‌دهد که در حال استفاد از روش **post** به منظور تسلیم داده فرم هستیم.

```
1 // Fig. 19.12: post.cpp
2 // Demonstrates POST method with XHTML form.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6
7 #include <string>
8 using std::string;
9
10 #include <cstdlib>
11 using std::getenv;
12 using std::atoi;
13
14 int main()
15 {
16    char postString[ 1024 ] = ""; // variable to hold POST data
17    string dataString = "";
18    string nameString = "";
19    string wordString = "";
20    int contentLength = 0;
21
22    // content was submitted
23    if ( getenv( "CONTENT_LENGTH" ) )
24    {
25       contentLength = atoi( getenv( "CONTENT_LENGTH" ) );
26       cin.read( postString, contentLength );
27       dataString = postString;
28    } // end if
29
30    cout << "Content-Type: text/html\n\n"; // output header
31
32    // output XML declaration and DOCTYPE
33    cout << "<?xml version = \"1.0\"?>"
34       << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.1//EN\" "
35       << "\"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd\">";
36
37    // output XHTML element and some of its contents
38    cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\">"
39       << "<head><title>Using POST with Forms</title></head><body>";
40
41    // output XHTML form
42    cout << "<p>Enter one of your favorite words here:</p>"
43       << "<form method = \"post\" action = \"post.cgi\">"
44       << "<input type = \"text\" name = \"word\" />"
45       << "<input type = \"submit\" value = \"Submit Word\" /></form>";
46
47    // data was sent using POST
```

```
48    if ( contentLength > 0 )
49    {
50       int nameLocation = dataString.find_first_of( "word=" ) + 5;
51       int endLocation = dataString.find_first_of( "&" ) - 1;
52
53       // retrieve entered word
54       wordString = dataString.substr(
55          nameLocation, endLocation - nameLocation );
56
57       if ( wordString == "" ) // no data was entered in text field
58          cout << "<p>Please enter a word.</p>";
59       else // output word
60          cout << "<p>Your word is: " << wordString << "</p>";
61    } // end if
62    else // no data was sent
63       cout << "<p>Please enter a word.</p>";
64
65    cout << "</body></html>";
66    return 0;
67 } // end main
```

شکل ۱۲-۱۹ | استفاده از روش post به همراه فرم XHTML.

سرویس‌دهنده وب مبادرت به ارسال داده **post** به اسکریپت CGI از طریق ورودی استاندارد می‌کند. داده همانند رشته **QUERY_STRING** کدگشایی می‌شود، اما متغیر محیطی **QUERY_STRING** تنظیم نمی‌شود. بجای آن، روش **post** اقدام به تنظیم متغیر محیطی **CONTENT_LENGTH** می‌کند تا نشاندهنده تعداد کاراکترهای داده باشد که همراه تقاضای post ارسال شده‌اند.

اسکریپت CGI از مقدار متغیر محیطی **CONTENT_LENGTH** برای پردازش حجم صحیحی از داده‌ها استفاده می‌کند. در اینصورت، خط 25 مقدار را خوانده و آنرا با فراخوانی تابع **atoi** تبدیل به مقدار صحیح (integer) می‌کند. خط 26 تابع **cin.read** را برای خواندن کاراکترها از ورودی استاندارد و ذخیره کاراکترها در آرایه **postString** فراخوانی می‌نماید. خط 27 داده **postString** را به یک رشته با تخصیص آن به **dataString** تبدیل می‌کند.

در فصل‌های اولیه، داده را از ورودی استاندارد و با استفاده از عبارتی همانند

```
cin >> data;
```

می‌خواندیم. همین روش در ارتباط با اسکریپت CGI کاربرد دارد که به همین منظور از عبارت **cin.read** استفاده کرده‌ایم. بخاطر دارید که **cin** داده را از ورودی استاندارد تا رسیدن به اولین کاراکتر خط جدید (newline)، فاصله یا tab هر کدام زودتر دیده شود، می‌خواند. ساختار CGI مستلزم افزوده شدن خط جدید پس از آخرین جفت نام–مقدار نیست. با اینکه برخی از مرورگرها یک خط جدید یا **EOF** الصاق می‌کنند، اما نیازی به انجام اینکار نیست. اگر **cin** با مرورگری بکار رود که فقط جفت‌های نام–مقدار را اضافه می‌کند، **cin** بایستی منتظر خط جدید باشد که هرگز نخواهد رسید. در چنین حالتی، سرویس‌دهنده شروع به شمارش زمان کرده و عاقبت اسکریپت CGI خاتمه می‌یابد. از اینرو، **cin.read** بر **cin** ترجیح داده می‌شود، چرا که برنامه‌نویس می‌تواند مقدار دقیق خواندن داده را تعیین کند.

۱۲-۱۹ سرآیندهای دیگر

یک اسکریپت CGI می‌تواند سرآیندهای دیگر HTTP را در کنار **Content-Type** بکار گیرد. در بسیاری از موارد، سرویس‌دهنده این سرآیندهای اضافی را به سرویس‌گیرنده ارسال می‌کند بدون اینکه آنها را اجرا کند. برای مثال، سرآیند **Refresh** در عبارت زیر مبادرت به هدایت سرویس‌گیرنده به مکان جدید پس از تعیین زمان مشخص می کند:

```
Referesh:"5: URL = http://www.deitel.com/newpage.html."
```

پنج ثانیه پس از اینکه مرورگر وب این سرآیند را دریافت کرد، مرورگر تقاضای منبع مشخص شده در URL را می‌کند. بطور جایگزین، سرآیند **Referesh** می‌تواند URL را نادیده بگیرد، که در اینحالت صفحه جاری پس از سپری شدن زمان، نوسازی می‌گردد.

ساختار CGI بر این نکته دلالت دارد که سرآیندها از نوع‌های خاص بجای آنکه مستقیماً به سرویس‌گیرنده ارسال شوند، توسط سرویس‌دهنده پردازش می‌شوند. اولین این سرآیندها، سرآیند **Location** است. همانند **Referesh**، این سرآیند، سرویس‌گیرنده را به مکان جدید هدایت می‌کند:

```
Location: http:/www.deitel.com/newpage.html
```

اگر به همراه یک URL نسبی (یا مجازی) بکار رود (یعنی Location:/newpage.html)، سرآیند **Location** به سرویس‌دهنده نشان می‌دهد که جهت حرکت در طرف سرویس‌دهنده انجام می‌شود بدون اینکه سرآیند **Location** به سرویس‌گیرنده برگردانده شود. در اینحالت، مستند بصورت راندو شده در مرورگر وب ظاهر می‌شود.

همچنین ساختار CGI شامل سرآیند **Status** است که به سرویس‌دهنده دستور می‌دهد تا خط وضعیت را بکار گیرد (همانند HTTP/1.1 200 OK). معمولاً سرویس‌دهنده خط وضعیت متناسب را به سرویس‌گیرنده ارسال می‌کند. با این همه، CGI به برنامه‌نویسان امکان داده تا تغییری در پاسخ وضعیت بوجود آورند. برای مثال، ارسال این سرآیند

```
status: 204 No Response
```

نشان می‌دهد که اگرچه تقاضا با موفقیت صورت گرفته، اما سرویس‌گیرنده، نمی‌تواند صفحه جدید را در پنجره مرورگر به نمایش در آورد.

به طور خلاصه CGI به اسکریپت‌ها اجازه می‌دهد تا با سرویس‌دهنده‌ها به سه روش در تعامل قرار گیرند:

۱- از طریق ارسال سرآیندها و محتویات به سرویس‌دهنده از طریق خروجی استاندارد.

۲- با تنظیم متغیرهای محیطی سرویس‌دهنده، که مقادیر آنها در درون اسکریپت قابل استفاده است.

۳- از طریق **POSTed**، کدگشایی داده URL که سرویس‌دهنده به ورودی استاندارد اسکریپت ارسال می‌کند.

۱۳-۱۹ مبحث آموزشی: صفحه وب تعاملی

برنامه شکل‌های ۱۳-۱۹ و ۱۴-۱۹ نحوه پیاده‌سازی یک *سردر*(*portal*) تعاملی برای وب سایت ساختگی Bug2Bug Travel را نشان می‌دهند. در این برنامه از سرویس‌گیرنده در ارتباط با نام و کلمه عبور سئوال شده، سپس اطلاعاتی در مورد سفر هفتگی براساس داده‌های وارد شده به نمایش در می‌آورد. برای ساده‌تر شدن کار، این مثال رمزگذاری بر روی داده‌های ارسالی به سرویس‌دهنده را انجام نمی‌دهد. بهتر است که داده‌های با اهمیت همانند کلمات عبور، حتماً رمزگذاری شوند. مبحث رمزگذاری خارج از قلمرو آموزشی این کتاب است.

برنامه شکل ۱۳-۱۹ صفحه آغازین را نشان می‌دهد. این صفحه یک مستند استاتیکی XHTML است که حاوی یک فرم بوده و داده‌ها را به اسکریپت CGI بنام **portal.cgi** پست می‌کند (خط 16). فرم حاوی یک فیلد برای دریافت نام کاربر (خط 18) و یکی برای دریافت کلمه عبور است (خط 19). [نکته: برخلاف اسکریپت‌های CGI که در شاخه **cgi-bin** سرویس‌دهنده وب جای داده می‌شوند، این مستند XHTML در شاخه **hotdocs** سرویس‌دهنده وب قرار داده شده است.]

```
1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
3     "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
4
5  <!-- Fig. 19.13: travel.html    -->
6  <!-- Bug2Bug Travel Homepage    -->
7
8  <html xmlns = "http://www.w3.org/1999/xhtml">
9     <head>
10       <title>Bug2Bug Travel</title>
11    </head>
12
13    <body>
14       <h1>Welcome to Bug2Bug Travel</h1>
15
16       <form method = "post" action = "/cgi-bin/portal.cgi">
17          <p>Please enter your name and password:</p>
18          <input type = "text" name = "namebox" />
19          <input type = "password" name = "passwordbox" />
20          <p>password is not encrypted</p>
21          <input type = "submit" name = "button" />
22       </form>
23    </body>
24 </html>
```

شکل ۱۳-۱۹ | سردر تعاملی برای ایجاد یک صفحه وب همراه با فیلد کلمه عبور.

برنامه شکل ۱۴-۱۹ حاوی اسکریپت CGI است. ابتدا، اجازه دهید تا به بررسی نحوه بازیابی نام و کلمه عبور کاربر از ورودی استاندارد و ذخیره آنها در رشته‌ها بپردازیم. تابع **find** از کلاس **string**، مبادرت به جستجوی **dataString** در خط 30 برای یافتن اولین پیشامد برای **namebox=** می‌کند. این تابع موقعیت قرارگیری **namebox=** را در رشته برگشت می‌دهد. برای بازیابی مقدار مرتبط با **namebox** یعنی مقدار

وارد شده توسط کاربر، مبادرت به انتقال موقعیت در رشته به میزان 8 کاراکتر به سمت جلو کرده‌ایم. اکنون برنامه حاوی یک مقدار صحیح است که به موقعیت شروع اشاره می‌کند. بخاطر دارید که رشته پرس‌وجو حاوی جفت‌های نام–مقدار است که توسط نمادهای تساوی و آمپرسنج (&) از یکدیگر متمایز می‌شوند. برای یافتن انتهای موقعیت داده، بدنبال کاراکتر & جستجوی انجام می‌دهیم (خط 31). طول کلمه وارد شده با عبارت محاسباتی **endNamelocation - namelocation** تعیین می‌شود. از روش مشابهی برای تعیین موقعیت شروع و پایانی کلمه عبور استفاده کرده‌ایم (خطوط 33-32). خطوط 38-36 مقادیر فیلدها را به متغیرهای **nameString** و **passwordString** تخصیص می‌دهند. از **nameString** در خط 52 برای چاپ یک پیغام خوش آمدگویی شخصی استفاده کرده‌ایم. سفر جاری توسط خطوط 66-53 به نمایش در می‌آید.

```
1  // Fig. 19.14: portal.cpp
2  // Handles entry to Bug2Bug Travel.
3  #include <iostream>
4  using std::cout;
5  using std::cin;
6
7  #include <string>
8  using std::string;
9
10 #include <cstdlib>
11 using std::getenv;
12 using std::atoi;
13
14 int main()
15 {
16    char postString[ 1024 ] = "";
17    string dataString = "";
18    string nameString = "";
19    string passwordString = "";
20    int contentLength = 0;
21
22    // data was posted
23    if ( getenv( "CONTENT_LENGTH" ) )
24       contentLength = atoi( getenv( "CONTENT_LENGTH" ) );
25
26    cin.read( postString, contentLength );
27    dataString = postString;
28
29    // search string for input data
30    int namelocation = dataString.find( "namebox=" ) + 8;
31    int endNamelocation = dataString.find( "&" );
32    int password = dataString.find( "passwordbox=" ) + 12;
33    int endPassword = dataString.find( "&button" );
34
35    // get values for name and password
36    nameString = dataString.substr(
37       namelocation, endNamelocation - namelocation );
38    passwordString = dataString.substr( password,endPassword - password );
39
40    cout << "Content-Type: text/html\n\n"; // output HTTP header
41
42    // output XML declaration and DOCTYPE
```

```
43    cout << "<?xml version = \"1.0\"?>"
44       << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.1//EN\" "
45       << "\"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd\">";
46
47    // output html element and some of its contents
48    cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\">"
49       << "<head><title>Bug2Bug Travel</title></head><body>";
50
51    // output specials
52    cout << "<h1>Welcome " << nameString << "!</h1>"
53       << "<p>Here are our weekly specials:</p>"
54       << "<ul><li>Boston to Taiwan ($875)</li>"
55       << "<li>San Diego to Hong Kong ($750)</li>"
56       << "<li>Chicago to Mexico City ($568)</li></ul>";
57
58    if ( passwordString == "coast2coast" ) // password is correct
59       cout << "<hr /><p>Current member special: "
60          << "Seattle to Tokyo ($400)</p>";
61    else // password was incorrect
62       cout << "<p>Sorry. You have entered an incorrect password</p>";
63
64    cout << "</body></html>";
65    return 0;
66 } // end main
```

شکل ۱۴-۱۹ | سردر تعاملی صفحه.

اگر کلمه عبور عضویت صحیح باشد، خطوط 60-59 موارد استثنایی و خاص‌ی را به نمایش در می‌آورند. اگر کلمه عبور اشتباه باشد، سرویس‌گیرنده مطلع می‌شود که کلمه عبور معتبر نیست و موارد استثنایی و خاص برای آن کاربر به نمایش در نمی‌آید.

توجه کنید که از یک صفحه استاتیک و یک اسکریپت CGI مجزا استفاده کرده‌ایم. البته می‌توانستیم هر دو کار را در یک اسکریپت CGI انجام دهیم.

۱۴-۱۹ کوکی

یکی از روش‌های پرکاربرد بهینه‌سازی تعامل‌های صورت گرفته با صفحات وب از طریق *Cookies* (*کوکی‌ها*) است. کوکی یک فایل متنی ذخیره شده توسط سایت وب بر روی هر کامپیوتر جداگانه است که به سایت اجازه می‌دهند تا فعالیت‌های بازدید کننده را ردگیری نماید. اولین باری که کاربر از سایت وب بازدید می‌کند، کامپیوتر کاربر یک فایل کوکی دریافت می‌کند، این کوکی هر بار که کاربر از آن سایت بازدید نماید، فعال می‌شود. اطلاعات جمع‌آوری شده بصورت یک رکورد بی‌نام هستند و حاوی داده‌ای می‌باشند که برای شخصی‌سازی محیط سایت بکار گرفته می‌شوند برای مثال، کوکی‌های موجود در برنامه‌های خرید، ممکن است هویت منحصر بفرد کاربران را ذخیره نمایند. هنگامی کاربر اقدام به افزودن ایتم‌های به کارت خرید *online* می‌کند یا اعمال دیگری انجام می‌دهد که نتیجه یک تقاضا از سرویس‌دهنده وب است، سرویس‌دهنده، کوکی را که حاوی اطلاعات منحصر بفرد کاربر است دریافت می‌نماید. سپس سرویس‌دهنده با استفاده از این اطلاعات، پردازش‌های مورد نیاز را انجام می‌دهد.

علاوه بر هویت بخشیدن به کاربران، کوکی‌ها می‌توانند دلالت بر سلایق مشتری‌ها نیز باشند. هنگامی یک برنامه وب، تقاضای از سوی یک سرویس‌گیرنده دریافت می‌کند، فرم وب می‌تواند اقدام به بررسی کوکی(هایی) ارسالی در دفعات قبل نماید و بلافاصله سلیقه مشتری را تشخیص داده و محصولات و سرویس‌های مطابق با آن سلیقه به نمایش در آورد.

به عنوان یک برنامه‌نویس، بایستی مطلع باشید که سرویس‌گیرنده‌ها می‌توانند کوکی‌ها را غیرفعال نمایند. در برنامه شکل‌های ۱۵-۱۹ الی ۱۷-۱۹ از کوکی‌ها برای ذخیره‌سازی و نگهداری اطلاعاتی در ارتباط با کاربر استفاده شده است.

```
1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
3     "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
4
5  <!-- Fig. 19.15: cookieform.html -->
6  <!-- Cookie Demonstration        -->
7
8  <html xmlns = "http://www.w3.org/1999/xhtml">
9     <head>
10       <title>Writing a cookie to the client computer</title>
11    </head>
12
13    <body>
14       <h1>Click Submit to save your cookie data.</h1>
15
16       <form method = "post" action = "/cgi-bin/writecookie.cgi">
17          <p>Name:<br />
18             <input type = "text" name = "name" />
19          </p>
20          <p>Age:<br />
21             <input type = "text" name = "age" />
22          </p>
23          <p>Favorite Color:<br />
24             <input type = "text" name = "color" />
25          </p>
26          <p>
27             <input type = "submit" name = "button" value="Submit"/>
28          </p>
29       </form>
30    </body>
31 </html>
```

شکل ۱۵-۱۹ | مستند XHTML حاوی فرمی برای پست داده به سرویس‌دهنده.

```
1  // Fig. 19.16: writecookie.cpp
2  // Program to write a cookie to a client's machine.
3  #include <iostream>
4  using std::cin;
5  using std::cout;
6
7  #include <string>
8  using std::string;
9
10 #include <cstdlib>
11 using std::getenv;
12 using std::atoi;
13
```

```
14 int main()
15 {
16    char query[ 1024 ] = "";
17    string dataString = "";
18    string nameString = "";
19    string ageString = "";
20    string colorString = "";
21    int contentLength = 0;
22
23    // expiration date of cookie
24    string expires = "Friday, 14-MAY-10 16:00:00 GMT";
25
26    // data was entered
27    if ( getenv( "CONTENT_LENGTH" ) )
28    {
29       contentLength = atoi( getenv( "CONTENT_LENGTH" ) );
30       cin.read(query,contentLength);//read data from standard input
31       dataString = query;
32
33       // search string for data and store locations
34       int nameLocation = dataString.find( "name=" ) + 5;
35       int endName = dataString.find( "&" );
36       int ageLocation = dataString.find( "age=" ) + 4;
37       int endAge = dataString.find( "&color" );
38       int colorLocation = dataString.find( "color=" ) + 6;
39       int endColor = dataString.find( "&button" );
40
41       // get value for user's name
42       nameString = dataString.substr(
43          nameLocation, endName - nameLocation );
44
45       if ( ageLocation > 0 ) // get value for user's age
46          ageString = dataString.substr(
47             ageLocation, endAge - ageLocation );
48
49       if ( colorLocation > 0 )// get value for user's favorite color
50          colorString = dataString.substr(
51             colorLocation, endColor - colorLocation );
52
53       // set cookie
54       cout << "Set-Cookie: Name=" << nameString << "age:"
55          << ageString << "color:" << colorString
56          << "; expires=" << expires << "; path=\n";
57    } // end if
58
59    cout << "Content-Type: text/html\n\n"; // output HTTP header
60
61    // output XML declaration and DOCTYPE
62    cout << "<?xml version = \"1.0\"?>"
63       << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.1//EN\" "
64       << "\"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd\">";
65
66    // output html element and some of its contents
67    cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\">"
68       << "<head><title>Cookie Saved</title></head><body>";
69
70    // output user's information
71    cout << "<p>A cookie has been set with the following"
72       << " data:</p><p>Name: " << nameString  << "<br/></p>"
73       << "<p>Age: " << ageString << "<br/></p>"
74       << "<p>Color: " << colorString << "<br/></p>"
75       << "<p>Click <a href=\"/cgi-bin/readcookie.cgi\">"
```

```
76       << "here</a> to read saved cookie data.</p></body></html>";
77    return 0;
78 } // end main
```

شکل ۱۶-۱۹ | نوشتن کوکی.

برنامه شکل ۱۵-۱۹ یک صفحه XHTML است که حاوی یک فرم بوده و مقادیری از طریق آن وارد می‌شوند. فرم مبادرت به ارسال اطلاعات به **writecookie.cgi** می‌کند (شکل ۱۶-۱۹). این اسکریپت CGI داده موجود در متغیر **CONTENT_LENGTH** را بازیابی می‌کند.

خط 24 از شکل ۱۶-۱۹ مبادرت به اعلان و مقداردهی اولیه رشته **expires** برای ذخیره‌سازی تاریخ انقضا کوکی می‌کند که تعیین کننده مدت زمانی است که کوکی می‌تواند بر روی ماشین سرویس‌گیرنده مقیم باشد. این مقدار می‌تواند یک رشته باشد، همانند مقداری که در این مثال بکار گرفته شده است، یا می‌تواند یک مقدار نسبی باشد. برای نمونه "**+30d**" مبادرت به تنظیم تاریخ انقضا کوکی پس از 30 روز می‌کند. برای برآورده کردن اهداف این فصل، تاریخ انقضا عمداً با سال 2010 تنظیم شده تا مطمئن گردیم که برنامه بخوبی درآیند اجرا خواهد شد. البته می‌توانید تاریخ انقضا در این مثال را به هر تاریخی که مایل هستید، تغییر دهید. پس از انقضا کوکی، مرورگر آنها را حذف خواهد کرد.

پس از بدست آوردن داده از فرم، برنامه یک کوکی ایجاد می‌کند (خطوط 56-54). در این مثال، یک کوکی ایجاد می‌کنیم که یک خط متنی حاوی جفت‌های نام–مقدار از داده پست شده که توسط کولن (:) از هم متمایز شده‌اند را ذخیره سازد. این خط بایستی قبل از اینکه سرآیند در سرویس‌گیرنده نوشته شود، خارج گردد. خط متنی با سرآیند **Set-Cookie:** آغاز می‌شود و نشان می‌دهد که مرورگر بایستی داده‌های ورودی را در یک کوکی ذخیره سازد. در این برنامه مبادرت به تنظیم سه صفت برای کوکی کرده‌ایم: یک جفت نام–مقدار حاوی داده که ذخیره خواهد شد، یک جفت نام–مقدار حاوی تاریخ انقضا و یک جفت نام–مقدار حاوی URL از دامنه سرویس‌دهنده (همانند www.deitel.com) برای اینکه کوکی معتبر باشد. در این مثال، **path** با هیچ مقداری تنظیم نشده است و از اینرو کوکی از طریق هر سرویس‌دهنده‌ای در دامنه سرویس‌دهنده که کوکی در آن نوشته شده است، قابل خواندن است. توجه کنید که جفت‌های نام–مقدار توسط سیمکولن از یکدیگر جدا شده‌اند. ما فقط از کاراکترهای کولن در درون داده کوکی خود استفاده کرده‌ایم تا تداخلی با قالب سرآیند **Set-Cookie:** بوجود نیاید. زمانیکه همان داده‌های به نمایش درآمده در شکل ۱۵-۱۹ را وارد کنیم، خطوط 56-54 داده "Name=Zoeage:24color:Red" را در کوکی ذخیره می‌کنند. خطوط 57-59 صفحه وبی ارسال می‌کنند تا نشان داده شود کوکی در سرویس‌گیرنده نوشته شده است.

برنامه شکل ۱۷-۱۹ کوکی نوشته شده در برنامه ۱۶-۱۹ را خوانده و اطلاعات ذخیره شده در آنرا به نمایش در می‌آورد. زمانیکه سرویس‌گیرنده تقاضای به سرویس‌دهنده ارسال می‌کند، مرورگر وب سرویس‌گیرنده بدنبال هر کوکی نوشته شده از سوی آن سرویس‌دهنده می‌گردد. این کوکی‌ها توسط مرورگر به سرویس‌دهنده بعنوان بخشی از تقاضا بازپس فرستاده می‌شوند. بر روی سرویس‌دهنده، متغیر محیطی **HTTP_COOKIE** کوکی‌های سرویس‌گیرنده را ذخیره می‌کند. خط 20 تابع **getenv** را با متغیر محیطی **HTTP_COOKIE** بعنوان پارامتر فراخوانی کرده و مقدار برگشتی را در **dataString** ذخیره می‌کند. جفت‌های نام–مقدار کدگشایی شده و رشته‌های مطابق با طرح کدگشایی بکار رفته در برنامه شکل ۱۶-۱۹ ذخیره می‌شوند (خط 24-23). خطوط 55-36 محتویات کوکی را در صفحه چاپ می‌کنند.

```
1  // Fig. 19.17: readcookie.cpp
2  // Program to read cookie data.
3  #include <iostream>
4  using std::cin;
5  using std::cout;
6
7  #include <string>
8  using std::string;
9
10 #include <cstdlib>
11 using std::getenv;
12
13 int main()
14 {
15    string dataString = "";
16    string nameString = "";
17    string ageString = "";
18    string colorString = "";
19
20    dataString = getenv( "HTTP_COOKIE" ); // get cookie data
21
22    // search through cookie data string
23    int nameLocation = dataString.find( "Name=" ) + 5;
24    int endName = dataString.find( "age:" );
25    int ageLocation = dataString.find( "age:" ) + 4;
26    int endAge = dataString.find( "color:" );
27    int colorLocation = dataString.find( "color:" ) + 6;
28
29    // store cookie data in strings
30    nameString = dataString.substr(
31       nameLocation, endName - nameLocation );
32    ageString = dataString.substr(
33       ageLocation, endAge - ageLocation );
34    colorString = dataString.substr( colorLocation );
35
36    cout << "Content-Type: text/html\n\n"; // output HTTP header
37
38    // output XML declaration and DOCTYPE
39    cout << "<?xml version = \"1.0\"?>"
40       << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.1//EN\" "
41       << "\"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd\">";
42
```

```
43    // output html element and some of its contents
44    cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\">"
45         << "<head><title>Read Cookies</title></head><body>";
46
47    if ( dataString != "" ) // data was found
48       cout << "<h3>The following data is saved in a cookie on"
49         <<" your computer</h3><p>Name:" << nameString <<"<br/></p>"
50          << "<p>Age: " << ageString << "<br/></p>"
51          << "<p>Color: " << colorString << "<br/></p>";
52    else // no data was found
53       cout << "<p>No cookie data.</p>";
54
55    cout << "</body></html>";
56    return 0;
57 } // end main
```

شکل ۱۷-۱۹ | برنامه، کوکی‌های ارسالی از کامپیوتر سرویس‌گیرنده را می‌خواند.

۱۵-۱۹ فایل‌های طرف سرویس‌گیرنده

در بخش قبلی، به توصیف نحوه نگهداری اطلاعات وضعیت که در ارتباط با کاربر هستند از طریق کوکی‌ها پرداختیم. مکانیزم‌های دیگری برای انجام اینکار وجود دارند که ایجاد فایلهای طرف سرویس‌گیرنده (*server-side files*) است، یعنی فایل‌های که بر روی سرویس‌گیرنده قرار داده می‌شوند یا بر روی شبکه سرویس‌دهنده. این روش تا حدی از امنیت بیشتر برخوردار است و مناسب نگهداری اطلاعات مهمتر است. در این مکانیزم، فقط یکنفر با داشتن مجوز دسترسی قادر به تغییر در فایل‌های موجود بر روی سرویس‌دهنده است. برنامه شکل‌های ۱۸-۱۹ و ۱۹-۱۹ از کاربران می‌خواهد تا اطلاعات تماس را وارد کرده، سپس آنرا بر روی سرویس‌دهنده ذخیره می‌کنند. شکل ۲۰-۱۹ فایلی را که توسط اسکریپت ایجاد شده، به نمایش در آورده است.

مستند XHTML در شکل ۱۸-۱۹ داده فرم را به اسکریپت CGI در شکل ۱۹-۱۹ پست می‌کند. در اسکریپت CGI، خطوط 92-45 پارامترهای که توسط سرویس‌گیرنده ارسال شده‌اند را کدگشایی می‌کنند. خط 105 نمونه‌ای از استریم فایل خروجی ایجاد می‌کند (*out File*) که فایلی را برای الصاق کردن باز می‌کند. اگر فایل **clients.txt** وجود نداشته باشد، آنرا ایجاد می‌کند. خطوط 116-114 اطلاعات شخصی را در فایل قرار می‌دهند.

```
1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
3     "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
4
5  <!-- Fig. 19.18: savefile.html          -->
6  <!-- Form to input client information   -->
7
8  <html xmlns = "http://www.w3.org/1999/xhtml">
9     <head>
10       <title>Please enter your contact information</title>
11    </head>
12
13    <body>
14       <p>Please enter your information in the form below.</p>
```

```
15       <p>Note: You must fill in all fields.</p>
16       <form method = "post" action = "/cgi-bin/savefile.cgi">
17          <p>
18             First Name:
19             <input type = "text" name = "firstname" size = "10" />
20             Last Name:
21             <input type = "text" name = "lastname" size = "15" />
22          </p>
23          <p>
24             Address:
25             <input type = "text" name = "address" size="25"/><br />
26             Town: <input type = "text" name = "town" size = "10" />
27             State: <input type = "text" name="state"size="2"/><br/>
28             Zip Code: <input type = "text" name="zipcode"size="5" />
29             Country: <input type = "text" name="country"size="10" />
30          </p>
31          <p>
32             E-mail Address: <input type = "text" name = "email" />
33          </p>
34             <input type = "submit" value = "Enter" />
35             <input type = "reset" value = "Clear" />
36       </form>
37    </body>
38 </html>
```

شکل ۱۸-۱۹ | مستند XHTML اطلاعات تماس کاربر را می‌خواند.

```
1  // Fig. 19.19: savefile.cpp
2  // Program to enter user's contact information into a
3  // server-side file.
4  #include <iostream>
5  using std::cerr;
6  using std::cin;
7  using std::cout;
8  using std::ios;
9
10 #include <fstream>
11 using std::ofstream;
12
13 #include <string>
14 using std::string;
15
16 #include <cstdlib>
17 using std::getenv;
18 using std::atoi;
19 using std::exit;
20
21 int main()
22 {
23    char postString[ 1024 ] = "";
24    int contentLength = 0;
25
26    // variables to store user data
27    string dataString = "";
28    string firstname = "";
29    string lastname = "";
30    string address = "";
31    string town = "";
32    string state = "";
33    string zipcode = "";
34    string country = "";
35    string email = "";
```

```
36
37    // data was posted
38    if ( getenv( "CONTENT_LENGTH" ) )
39       contentLength = atoi( getenv( "CONTENT_LENGTH" ) );
40
41    cin.read( postString, contentLength );
42    dataString = postString;
43
44    // search for first '+' character
45    string::size_type charLocation = dataString.find( "+" );
46
47    // search for next '+' character
48    while ( charLocation < string::npos )
49    {
50       dataString.replace( charLocation, 1, " " );
51       charLocation = dataString.find( "+", charLocation + 1 );
52    } // end while
53
54    // find location of firstname
55    int firstStart = dataString.find( "firstname=" ) + 10;
56    int endFirst = dataString.find( "&lastname" );
57    firstname = dataString.substr(firstStart,endFirst - firstStart );
58
59    // find location of lastname
60    int lastStart = dataString.find( "lastname=" ) + 9;
61    int endLast = dataString.find( "&address" );
62    lastname = dataString.substr( lastStart, endLast - lastStart );
63
64    // find location of address
65    int addressStart = dataString.find( "address=" ) + 8;
66    int endAddress = dataString.find( "&town" );
67    address=dataString.substr(addressStart,endAddress-addressStart );
68
69    // find location of town
70    int townStart = dataString.find( "town=" ) + 5;
71    int endTown = dataString.find( "&state" );
72    town = dataString.substr( townStart, endTown - townStart );
73
74    // find location of state
75    int stateStart = dataString.find( "state=" ) + 6;
76    int endState = dataString.find( "&zipcode" );
77    state = dataString.substr( stateStart, endState - stateStart );
78
79    // find location of zip code
80    int zipStart = dataString.find( "zipcode=" ) + 8;
81    int endZip = dataString.find( "&country" );
82    zipcode = dataString.substr( zipStart, endZip - zipStart );
83
84    // find location of country
85    int countryStart = dataString.find( "country=" ) + 8;
86    int endCountry = dataString.find( "&email" );
87    country = dataString.substr( countryStart, endCountry - countryStart );
88
89    // find location of e-mail address
90    int emailStart = dataString.find( "email=" ) + 6;
91    int endEmail = dataString.find( "&submit" );
92    email = dataString.substr( emailStart, endEmail - emailStart );
93
94    cout << "Content-Type: text/html\n\n"; // output header
95
96    // output XML declaration and DOCTYPE
97    cout << "<?xml version = \"1.0\"?>"
```

```
98       << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.1//EN\" "
99       << "\"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd\">";
100
101   // output html element and some of its contents
102    cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\">"
103       << "<head><title>Contact Information entered</title></head><body>";
104
105    ofstream outFile( "clients.txt", ios::app ); // output to file
106
107    if ( !outFile ) // file was not opened properly
108    {
109       cerr << "Error: could not open contact file.";
110       exit( 1 );
111    } // end if
112
113    // append data to clients.txt file
114    outFile <<firstname << " " << lastname << "\n" <<address << "\n"
115       << town << " " << state << " " << country << " " << zipcode
116       << "\n" << email << "\n\n";
117
118    // output data to user
119    cout << "<table><tbody><tr><td>First Name:</td><td>" << firstname
120       << "</td></tr><tr><td>Last Name:</td><td>" << lastname
121       << "</td></tr><tr><td>Address:</td><td>" << address
122       << "</td></tr><tr><td>Town:</td><td>" << town
123       << "</td></tr><tr><td>State:</td><td>" << state
124       << "</td></tr><tr><td>Zip Code:</td><td>" << zipcode
125       << "</td></tr><tr><td>Country:</td><td>" << country
126       << "</td></tr><tr><td>Email:</td><td>" << email
127       << "</td></tr></tbody></table></body>\n</html>\n";
128    return 0;
129 } // end main
```

شکل ۱۹-۱۹ | ایجاد فایل طرف سرویس‌دهنده برای ذخیره اطلاعات کاربر.

```
Jane Doe
123 Main Street
Boston MA USA 12345
jan@doe.com
```

شکل ۲۰-۱۹ | محتویات فایل clients.txt.

چند نکته با اهمیت در ارتباط با این برنامه وجود دارد. اول اینکه، هیچ گونه عملیات اعتبارسنجی داده، قبل از نوشتن آنها بر روی دیسک انجام نداده‌ایم. معمولاً، بایستی اسکریپت به بررسی صحت داده‌ها بپردازد. دوم اینکه، فایل ما در شاخه **cgi-bin** قرار دارد که در دسترس عموم است. هر کسی که نام فایل را بداند می‌تواند به آسانی آنرا پیدا کرده و به محتویات دسترسی دسترسی داشته باشد.

این اسکریپت بقدر کافی از کفایت عرضه بر روی اینترنت برخوردار نیست، اما مثالی از نحوه استفاده از فایلهای طرف سرویس‌گیرنده به منظور ذخیره‌سازی اطلاعات است. از آنجا که فایل‌ها بر روی سرویس‌دهنده ذخیره می‌شوند، کاربران نمی‌تواند آنها را تغییر دهند مگر اینکه مجوز انجام اینکار را از مدیر سرویس‌دهنده کسب کرده باشند. بنابر این ذخیره این فایل‌ها بر روی سرویس‌دهنده امن‌تر از

ذخیره‌سازی داده‌های کاربر در کوکی‌ها است. [نکته: برخی از سیستم‌ها اطلاعات کاربر را در پایگاه داده‌های حفاظت شده ذخیره می‌کنند که از سطح امنیتی بالاتری برخوردار است.]

در این بخش با نحوه نوشتن داده در یک فایل طرف سرویس‌گیرنده آشنا شدید. در بخش بعدی شما را با نحوه بازیابی داده‌ها از فایل طرف سرویس‌گیرنده با استفاده از تکنیک‌های معرفی شده در فصل هفدهم آشنا خواهیم کرد.

۱۹-۱۶ مبحث آموزشی کارت خرید

بسیاری از وب سایت‌های تجاری دارای برنامه‌های کاربردی کارت خرید هستند که به مشتریان امکان خرید راحت ایتم‌هایی از وب را فراهم می‌آورند. این سایت‌ها هر آنچه که مشتری می‌خواهد خرید کند ثبت کرده و روش خرید online آسانی در اختیار وی قرار می‌دهند. سپس مشتریان با استفاده از یک کارت خرید الکترونیکی اقدام به خرید می‌کنند، همانطوری که از یک فروشگاه عادی خرید می‌نمایند. همانطوری که کاربران اقدام به افزودن آیتم‌های مورد نظر خود به کارت خرید می‌کنند، سایت محتویات کارت را به روز می‌نماید. پس از تایید کاربر (مشتری)، هزینه آیتم‌ها از کارت خرید دریافت می‌شود. برای آشنایی با تجارت الکترونیکی و کارت خرید در دنیای واقعی، پیشنهاد می‌کنیم تا سری به کتابفروشی Amazan به آدرس www.amazon.com بزنید.

کارت خرید که در این بخش پیاده‌سازی می‌شود (شکل‌های ۲۱-۱۹ الی ۲۴-۱۹) به کاربران امکان خرید کتاب‌های را از یک کتابفروشی فرضی که فقط چهار کتاب می‌فروشد را می‌دهد (به شکل ۲۳-۱۹ نگاه کنید). در این مثال، از چهار اسکریپت، دو فایل طرف سرویس‌دهنده و کوکی‌ها استفاده شده است.

برنامه شکل ۲۱-۱۹ اولین اسکریپت از چهار اسکریپت یاد شده است، که صفحه login (ورود) است. این اسکریپت از تمام اسکریپت‌های این بخش پیچیده‌تر است.

```
1   // Fig. 19.21: login.cpp
2   // Program to output an XHTML form, verify the
3   // username and password entered, and add members.
4   #include <iostream>
5   using std::cerr;
6   using std::cin;
7   using std::cout;
8   using std::ios;
9
10  #include <fstream>
11  using std::fstream;
12
13  #include <string>
14  using std::string;
15
16  #include <cstdlib>
17  using std::getenv;
18  using std::atoi;
19  using std::exit;
20
```

```
21  void header();
22  void writeCookie();
23
24  int main()
25  {
26     char query[ 1024 ] = "";
27     string dataString = "";
28
29     // strings to store username and password
30     string userName = "";
31     string passWord = "";
32
33     int contentLength = 0;
34     bool newMember = false;
35
36     // data was posted
37     if ( getenv( "CONTENT_LENGTH" ) )
38     {
39        // retrieve query string
40        contentLength = atoi( getenv( "CONTENT_LENGTH" ) );
41        cin.read( query, contentLength );
42        dataString = query;
43
44        // find username location
45        int userLocation = dataString.find( "user=" ) + 5;
46        int endUser = dataString.find( "&" );
47
48        // find password location
49        int passwordLocation = dataString.find( "password=" ) + 9;
50        int endPassword = dataString.find( "&new" );
51
52        if ( endPassword > 0 ) // new membership requested
53        {
54           newMember = true;
55           passWord = dataString.substr(
56              passwordLocation, endPassword - passwordLocation );
57        } // end if
58        else // existing member
59           passWord = dataString.substr( passwordLocation );
60
61        userName = dataString.substr(
62           userLocation, endUser - userLocation );
63     } // end if
64
65     // no data was retrieved
66     if ( dataString == "" )
67     {
68        header();
69        cout << "<p>Please login.</p>";
70
71        // output login form
72        cout << "<form method = \"post\" action = \"/cgi-bin/login.cgi\">"
73           << "<p>User Name: <input type = \"text\" name = \"user\"/><br/>"
74           << "Password: <input type = \"password\" name = \"password\"/>"
75           << "<br/>New? <input type = \"checkbox\" name = \"new\""
76           << " value = \"1\"/></p>"
77           << "<input type = \"submit\" value = \"login\"/></form>";
78     } // end if
79     else // process entered data
80     {
81        string fileUsername = "";
82        string filePassword = "";
```

```
83          bool userFound = false;
84
85          // open user data file for reading and writing
86          fstream userData( "userdata.txt", ios::in | ios::out);
87
88          if ( !userData ) // could not open file
89          {
90             cerr << "Could not open database.";
91             exit( 1 );
92          } // end if
93
94          // add new member
95          if ( newMember )
96          {
97             // read username and password from file
98             while ( !userFound && userData >> fileUsername >> filePassword )
99             {
100               if ( userName == fileUsername ) // name is already taken
101                  userFound = true;
102            } // end while
103
104            if ( userFound ) // user name is taken
105            {
106               header();
107               cout << "<p>This name has already been taken.</p>"
108                  << "<a href=\"/cgi-bin/login.cgi\">Try Again</a>";
109            } // end if
110            else // process data
111            {
112               writeCookie(); // write cookie
113               header();
114
115               // write user data to file
116               userData.clear(); //clear eof, allow write at end of file
117               userData << "\n" << userName << "\n" << passWord;
118
119               cout << "<p>Your information has been processed."
120                  << "<a href=\"/cgi-bin/shop.cgi\">Start Shopping</a></p>";
121            } // end else
122         } // end if
123         else // search for password if entered
124         {
125            bool authenticated = false;
126
127            // read in user data
128            while ( !userFound && userData >> fileUsername >> filePassword )
129            {
130               // username was found
131               if ( userName == fileUsername )
132               {
133                  userFound = true;
134
135                  // determine whether password is correct
136                  // and assign bool result to authenticated
137                  authenticated = ( passWord == filePassword );
138               } // end if
139            } // end while
140
141            // user is authenticated
142            if ( authenticated )
143            {
144               writeCookie();
```

```
145             header();
146
147             cout << "<p>Thank you for returning, " << userName << "!</p>"
148                << "<a href=\"/cgi-bin/shop.cgi\">Start Shopping</a>";
149          } // end if
150          else // user not authenticated
151          {
152             header();
153
154             if ( userFound ) // password is incorrect
155                cout << "<p>You have entered an incorrect password. "
156                   << "Please try again.</p>"
157                   << "<a href=\"/cgi-bin/login.cgi\">Back to login</a>";
158             else // user is not registered
159                cout << "<p>You are not a registered user.</p>"
160                   << "<a href=\"/cgi-bin/login.cgi\">Register</a>";
161          } // end else
162       } // end else
163    } // end else
164
165    cout << "</body>\n</html>\n";
166    return 0;
167 } // end main
168
169 // function to output header
170 void header()
171 {
172    cout << "Content-Type: text/html\n\n"; // output header
173
174    // output XML declaration and DOCTYPE
175    cout << "<?xml version = \"1.0\"?>"
176       << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.1//EN\" "
177       << "\"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd\">";
178
179    // output html element and some of its contents
180    cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\">"
181       << "<head><title>Login Page</title></head><body>";
182 } // end function header
183
184 // function to write cookie data
185 void writeCookie()
186 {
187    string expires = "Friday, 14-MAY-10 16:00:00 GMT";
188    cout << "Set-Cookie: CART=; expires=" << expires << "; path=\n";
189 } // end function writeCookie
```

شکل ۲۱-۱۹ | برنامه‌ای که صفحه login را بوجود می‌آورد.

اولین شرط **if** در خط 37 تعیین می‌کند که آیا داده به برنامه ارسال شده است یا خیر. دومین شرط **if** (خط 66) تعیین می‌کند که آیا **dataString** تهی مانده است یا خیر (یعنی داده برای کدگشایی تسلیم نشده یا کدگشایی با موفقیت صورت نگرفته است). اولین بار که این برنامه را اجرا می‌کنیم، شرط اول برقرار نیست و شرط دوم برقرار است، از اینرو خطوط 77-72 فرم XHTML را در اختیار کاربر قرار می‌دهند، همانند اولین تصویر در شکل ۲۱-۱۹. زمانیکه کاربر فرم را پر کرده و بر روی دکمه **login** کلیک کند، فایل **login.cgi** مجدداً تقاضا می‌شود، این بار تقاضا حاوی داده پست شده است، از اینرو شرط موجود در خط 37 با **true** ارزیابی شده و شرط موجود در خط 66 با **false** ارزیابی می‌شود.

اگر کاربر داده را تسلیم کند، کنترل برنامه با بلوک **else** که از خط 79 شروع می‌شود، ادامه می‌یابد، مکانی که اسکریپت داده را پردازش می‌کند. خط 86 فایل **userdata.txt** را باز می‌کند، فایلی که حاوی کلیه اسامی کاربران و کلمات عبور برای اعضای موجود است. اگر کاربر جعبه‌چک **New** را انتخاب کند تا یک عضویت جدید ایجاد شود، شرط موجود در خط 95 با **true** ارزیابی شده و اسکریپت مبادرت به ثبت اطلاعات کاربر در فایل **userdata.txt** در سرویس‌دهنده می‌کند. خطوط 102-98 این فایل را خوانده، نام کاربر را با نام وارد شده مقایسه می‌کنند.

اگر نام کاربر در فایل از قبل وجود داشته باشد، حلقه موجود در خطوط 102-98 قبل از رسیدن به انتهای فایل خاتمه می‌یابد و خطوط 108-107 پیغام مناسبی در اختیار کاربر قرار داده و یک فوق لینک وی را به فرم باز می‌گرداند. اگر نام کاربری وارد شده در فایل **user.data** وجود نداشته باشد، خط 117 اطلاعات کاربر جدید را به فایل با فرمت

```
Bernard
blue
```

اضافه می‌کند. هر نام کاربری و کلمه عبور توسط یک کاراکتر خط جدید از هم متمایز می‌شوند. خطوط 120-119 یک فوق لینک به اسکریپت برنامه شکل ۲۲-۱۹ فراهم می‌آورند که به کاربران امکان خرید را می‌دهد.

آخرین سناریو ممکنه برای این اسکریپت برگشت دادن کاربران است (خطوط 162-123). این بخش از برنامه زمانی اجرا می‌شود که کاربر، نام و کلمه عبور را وارد کند، اما جعبه‌چک **New** را انتخاب نکند. در اینحالت، فرض می‌کنیم که کاربر در حال حاضر دارای یک نام کاربری و کلمه عبور در فایل **userdata.txt** است. خطوط 139-128 کل **userdata.txt** را خوانده و اقدام به یافتن نام کاربر وارد شده می‌کنند. اگر نام کاربر پیدا شود (خط 131)، تعیین می‌کنیم که آیا کلمه عبور وارد شده مطابق با کلمه عبور ذخیره شده در فایل است یا خیر (خط 137). اگر چنین باشد، متغیر بولی **authenticated** با **true** تنظیم می‌شود. در غیر اینصورت با **false** تنظیم می‌گردد. اگر هویت کاربر تایید شود (خط 142)، خط 144 تابع **writeCookie** را برای مقداردهی اولیه یک کوکی بنام **CART** فراخوانی می‌کند (خط 188) که توسط اسکریپت‌های دیگر برای ذخیره‌سازی داده مرتبط با کتاب‌های انتخابی توسط کاربر که به کارت خرید افزوده می‌شوند، بکار گرفته می‌شود. توجه کنید که این کوکی جایگزین هر کوکی موجود همنام شده و داده موجود از جلسه قبل از بین می‌رود. پس از ایجاد کوکی، اسکریپت پیغام خوش آمدگویی به کاربر را به نمایش درآورده و لینکی به **shop.cgi** فراهم می‌آورد که کاربر می‌تواند از آنجا اقدام به خرید کتاب کند (خطوط 148-147).

اگر کاربر تایید نشود، برنامه دلیل آن را مشخص می‌کند (خطوط 160-154). اگر کاربر پیدا شود اما تایید نشود، پیغامی به نمایش در آمده و نشان می‌دهد که کلمه عبور معتبر نبوده است (خطوط 157-155). یک فوق لینک برای صفحه **login** در نظر گرفته شده است که کاربر می‌تواند دوباره از آن طریق اقدام کند.

اگر نام کاربری و هم کلمه عبور هر دو پیدا نشوند، پس یک کاربر ثبت نشده اقدام به ورود کرده است. خطوط 160-159 پیغام مبنی بر اینکه کاربر دارای مجوزهای صحیح برای دسترسی به صفحه نیست به نمایش درآورده و لینکی فراهم می‌آورند که کاربر بتواند دوباره اقدام به ورود کند.

برنامه شکل ۲۲-۱۹ از مقادیر موجود در **catalog.txt** برای چاپ اطلاعات در یک جدول XHTML استفاده کرده (شکل ۲۵-۱۹) است، آیتم‌های که کاربر می‌تواند خرید کند (خطوط 82-45). ستون آخر در هر سطر شامل یک دکمه برای افزودن آن آیتم به کارت خرید است. خطوط 65-63 مقادیر مختلف برای هر کتاب را چاپ کرده و خطوط 76-71 فرمی حاوی دکمه **submit** را برای افزودن هر کتاب به کارت خرید فراهم می‌آورند. فیلدهای پنهان فرم خاص هر کتاب بوده و مرتبط با اطلاعات آن کتاب هستند. توجه کنید که نتیجه مستند XHTML به سرویس‌گیرنده حاوی چند فرم، یکی برای هر کتاب ارسال می‌شود. با این همه، کاربر می‌تواند فقط در هر بار یک فرم را تسلیم (submit) کند. جفت‌های نام–مقدار فیلدهای پنهان در میان فرم تسلیم شده به اسکریپت **viewcart.cgi** ارسال می‌شوند.

```
1  // Fig. 19.22: shop.cpp
2  // Program to display available books.
3  #include <iostream>
4  using std::cerr;
5  using std::cout;
6  using std::ios;
7
8  #include <fstream>
9  using std::ifstream;
10
11 #include <string>
12 using std::string;
13
14 #include <cstdlib>
15 using std::exit;
16
17 void header();
18
19 int main()
20 {
21    // variables to store product information
22    char book[ 50 ] = "";
23    char year[ 50 ] = "";
24    char isbn[ 50 ] = "";
25    char price[ 50 ] = "";
26
27    string bookString = "";
28    string yearString = "";
29    string isbnString = "";
30    string priceString = "";
31
```

```
32    ifstream userData( "catalog.txt", ios::in ); // open file for input
33
34    // file could not be opened
35    if ( !userData )
36    {
37       cerr << "Could not open database.";
38       exit( 1 );
39    } // end if
40
41    header(); // output header
42
43    // output available books
44    cout << "<center><br/>Books available for sale<br/><br/>"
45       << "<table border = \"1\" cellpadding = \"7\" >";
46
47    // file is open
48    while ( userData )
49    {
50       // retrieve data from file
51       userData.getline( book, 50 );
52       bookString = book;
53
54       userData.getline( year, 50 );
55       yearString = year;
56
57       userData.getline( isbn, 50 );
58       isbnString = isbn;
59
60       userData.getline( price, 50 );
61       priceString = price;
62
63       cout << "<tr><td>" << bookString << "</td><td>" << yearString
64          << "</td><td>" << isbnString << "</td><td>" << priceString
65          << "</td>";
66
67       // file is still open after reads
68       if ( userData )
69       {
70          // output form with buy button
71          cout << "<td><form method=\"post\" "
72             << "action=\"/cgi-bin/viewcart.cgi\">"
73             << "<input type=\"hidden\" name=\"add\" value=\"true\"/>"
74             << "<input type=\"hidden\" name=\"isbn\" value=\""
75             << isbnString << "\"/>" << "<input type=\"submit\" "
76             << "value=\"Add to Cart\"/>\n</form></td>\n";
77       } // end if
78
79       cout << "</tr>\n";
80    } // end while
81
82    cout << "</table></center><br/>"
83       << "<a href=\"/cgi-bin/checkout.cgi\">Check Out</a>"
84       << "</body></html>";
85    return 0;
86 } // end main
87
88 // function to output header information
89 void header()
90 {
91    cout << "Content-Type: text/html\n\n"; // output header
92
93    // output XML declaration and DOCTYPE
```

```
94    cout << "<?xml version = \"1.0\"?>"
95       << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.1//EN\" "
96       << "\"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd\">";
97
98    // output html element and some of its contents
99    cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\">"
100          << "<head><title>Shop Page</title></head><body>";
101    } // end function header
```

شکل ۲۲-۱۹ | اسکریپت CGI که به کاربران امکان خرید کتاب را می‌دهد.

پس از خرید یک کتاب توسط کاربر، اسکریپت **viewcart.cgi** تقاضا شده و ISBN کتاب خریداری شده به اسکریپت از طریق یک فیلد پنهان در فرم ارسال می‌شود. شکل ۲۳-۱۹ با خواندن مقدار از کوکی ذخیره شده بر روی سیستم کاربر آغاز بکار می‌کند (خط 35). هر داده کوکی موجود در رشته **cookieString** ذخیره شده است (خط 36). عدد ISBN وارد شده از فرم در شکل ۲۲-۱۹ در رشته **isbnEntered** ذخیره می‌شود (خط 52). سپس اسکریپت تعیین می‌کند که آیا کارت در حال حاضر حاوی داده می‌باشد یا خیر (خط 61). اگر نباشد، رشته **cookieString** مقدار وارد شده ISBN را دریافت می‌کند (خط 62). اگر کوکی در حال حاضر حاوی داده باشد، ISBN وارد شده به داده کوکی موجود الصاق می‌شود (خط 64). کتاب جدید در کوکی **CART** در خطوط 68-67 ذخیره می‌شود. خط 84 محتوی کارت را در جدولی با فراخوانی تابع **displayShoppingCart** به نمایش در می آورد.

تابع **displayShoopingCart** ایتم‌های موجود در کارت خرید را در یک جدول به نمایش در می‌آورد. خط 109 فایل طرف سرویس‌دهنده بنام **catalog.txt** را باز می‌کند. اگر فایل با موفقیت باز شود، خطوط 155-122 اطلاعات هر کتاب را از فایل دریافت می‌کنند. خطوط 138-125 این اطلاعات را در شی‌های رشته‌ای ذخیره می‌کنند. خطوط 148-140 تعداد دفعاتی که ISBN جاری در کوکی ظاهر شده است را می‌شمارند (یعنی کارت خرید). اگر کتاب جاری در کارت کاربر دیده شود، خطوط 154-151 یک سطر جدول حاوی عنوان کتاب، کپی رایت، ISBN و قیمت را به همراه تعداد کتاب درخواستی را به نمایش در می‌آورند.

```
1   // Fig. 19.23: viewcart.cpp
2   // Program to view books in the shopping cart.
3   #include <iostream>
4   using std::cerr;
5   using std::cin;
6   using std::cout;
7   using std::ios;
8
9   #include <fstream>
10  using std::ifstream;
11
12  #include <string>
13  using std::string;
14
15  #include <cstdlib>
16  using std::getenv;
```

```
17  using std::atoi;
18  using std::exit;
19
20  void displayShoppingCart( const string & );
21
22  int main()
23  {
24     char query[ 1024 ] = ""; // variable to store query string
25     string cartData; // variable to hold contents of cart
26
27     string dataString = "";
28     string cookieString = "";
29     string isbnEntered = "";
30     int contentLength = 0;
31
32     // retrieve cookie data
33     if ( getenv( "HTTP_COOKIE" ) )
34     {
35        cartData = getenv( "HTTP_COOKIE" );
36        cookieString = cartData;
37     } // end if
38
39     // data was entered
40     if ( getenv( "CONTENT_LENGTH" ) )
41     {
42        contentLength = atoi( getenv( "CONTENT_LENGTH" ) );
43        cin.read( query, contentLength );
44        dataString = query;
45
46        // find location of isbn value
47        int addLocation = dataString.find( "add=" ) + 4;
48        int endAdd = dataString.find( "&isbn" );
49        int isbnLocation = dataString.find( "isbn=" ) + 5;
50
51        // retrieve isbn number to add to cart
52        isbnEntered = dataString.substr( isbnLocation );
53
54        // write cookie
55        string expires = "Friday, 14-MAY-10 16:00:00 GMT";
56        int cartLocation = cookieString.find( "CART=" ) + 5;
57
58        if ( cartLocation > 4 ) // cookie exists
59           cookieString = cookieString.substr( cartLocation );
60
61        if ( cookieString == "" ) // no cookie data exists
62           cookieString = isbnEntered;
63        else // cookie data exists
64           cookieString += "," + isbnEntered;
65
66        // set cookie
67        cout << "Set-Cookie: CART=" << cookieString << "; expires="
68           << expires << "; path=\n";
69     } // end if
70
71     cout << "Content-Type: text/html\n\n"; // output HTTP header
72
73     // output XML declaration and DOCTYPE
74     cout << "<?xml version = \"1.0\"?>"
75        << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.1//EN\" "
76        << "\"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd\">";
77
78     // output html element and some of its contents
```

```
79     cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\">"
80        << "<head><title>Shopping Cart</title></head>"
81        << "<body><center><p>Here is your current order:</p>";
82
83     if ( cookieString != "" ) // cookie data exists
84        displayShoppingCart( cookieString );
85     else
86        cout << "The shopping cart is empty.";
87
88     // output links back to book list and to check out
89     cout << "</center><br/>";
90     cout << "<a href=\"/cgi-bin/shop.cgi\">Back to book list</a><br/>";
91     cout << "<a href=\"/cgi-bin/checkout.cgi\">Check Out</a>";
92     cout << "</body></html>\n";
93     return 0;
94  } // end main
95
96  // function to display items in shopping cart
97  void displayShoppingCart( const string &cookieRef )
98  {
99     char book[ 50 ] = "";
100    char year[ 50 ] = "";
101    char isbn[ 50 ] = "";
102    char price[ 50 ] = "";
103
104    string bookString = "";
105    string yearString = "";
106    string isbnString = "";
107    string priceString = "";
108
109    ifstream userData( "catalog.txt", ios::in ); // open file for input
110
111    if ( !userData ) // file could not be opened
112    {
113       cerr << "Could not open database.";
114       exit( 1 );
115    } // end if
116
117    cout << "<table border = 1 cellpadding = 7 >";
118    cout << "<tr><td>Title</td><td>Copyright</td><td>ISBN</td>"
119       << "<td>Price</td><td>Count</td></tr>";
120
121    // file is open
122    while ( !userData.eof() )
123    {
124       // retrieve book information
125       userData.getline( book, 50 );
126       bookString = book;
127
128       // retrieve year information
129       userData.getline( year, 50 );
130       yearString = year;
131
132       // retrieve isbn number
133       userData.getline( isbn, 50 );
134       isbnString = isbn;
135
136       // retrieve price
137       userData.getline( price, 50 );
138       priceString = price;
139
140       int match = cookieRef.find( isbnString, 0 );
```

```
141       int count = 0;
142
143       // match has been made
144       while ( match >= 0 && isbnString != "" )
145       {
146          count++;
147          match = cookieRef.find( isbnString, match + 13 );
148       } // end while
149
150       // output table row with book information
151       if ( count != 0 )
152          cout << "<tr><td>" << bookString << "</td><td>" << yearString
153             << "</td><td>" << isbnString << "</td><td>" << priceString
154             << "</td><td>" << count << "</td></tr>";
155    } // end while
156
157    cout << "</table>"; // end table
158 } // end function displayShoppingCart
```

شکل ۲۳-۱۹ | اسکریپت CGI که به کاربران امکان مشاهده محتویات کارت خرید را می‌دهد.

برنامه شکل ۲۴-۱۹ صفحه‌ای است که به هنگام انتخاب لینک **check out** (یعنی خرید کتاب‌های موجود در کتاب خرید) به نمایش در می‌آید. این اسکریپت پیغامی به کاربر نشان داده و تابع writeCookie را فراخوانی می‌کند (خط 13) که اطلاعات جاری در کارت خرید را پاک می‌کند. شکل ۲۵-۱۹ نشان دهنده محتویات فایل **catalog.txt** است. این فایل بایستی در همان شاخه‌ای باشد که اسکریپت‌های CGI قرار دارند.

```
1   // Fig. 19.24: checkout.cpp
2   // Program to log out of the system.
3   #include <iostream>
4   using std::cout;
5
6   #include <string>
7   using std::string;
8
9   void writeCookie();
10
11  int main()
12  {
13     writeCookie(); // write the cookie
14     cout << "Content-Type: text/html\n\n"; // output header
15
16     // output XML declaration and DOCTYPE
17     cout << "<?xml version = \"1.0\"?>"
18        << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.1//EN\" "
19        << "\"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd\">";
20
21     // output html element and its contents
22     cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\">"
23        << "<head><title>Checked Out</title></head><body><center>"
24        << "<p>You have checked out<br />"
25        << "You will be billed accordingly<br />To login again, "
26        << "<a href=\"/cgi-bin/login.cgi\">click here</a>"
27        << "</center></body></html>\n";
28     return 0;
29  } // end main
30
```

```
31  // function to write cookie
32  void writeCookie()
33  {
34     // string containing expiration date
35     string expires = "Friday, 14-MAY-10 16:00:00 GMT";
36
37     // set cookie
38     cout << "Set-Cookie: CART=; expires=" << expires << "; path=\n";
39  } // end writeCookie
```

شکل ۲۴-۱۹ برنامه تصفیه حساب.

```
visual Basic .NET How to Program
2002
0-13-029363-6
$50.00
C# How to Program
2002
0-13-062221-4
$49.95
C How to Program 4e
2004
0-13-142644-3
$88.00
Java How to Program 6e
2005
0-13-148398-6
$88.00
```

شکل ۲۵-۱۹ محتویات فایل catalog.txt

منابع اینترنت و وب

Apache

```
httpd.apache.org
www.apacheweek.com
linuxtoday.com/stories/18780.html
```

CGI

```
www.gnu.org/software/cgicc/cgicc.html
www.hotscripts.com
www.jmarshall.com/easy/cgi
www.w3.org/CGI
www.w3.org/Protocols
```

فصل بیستم

جستجو و مرتب‌سازی

اهداف

- جستجو برای یافتن مقداری در یک بردار با استفاده از روش جستجوی باینری.
- مرتب‌سازی بردار با استفاده از الگوریتم بازگشتی بازگشتی merge.
- تعیین کارایی الگوریتم‌های جستجو و مرتب‌سازی.

رئوس مطالب

20-1 مقدمه

جستجوی داده‌ها مستلزم تعیین وجود یک مقدار (که از آن به عنوان کلید جستجو یاد می‌شود) در میان مقادیر دیگر بوده و اگر چنین باشد، موقعیت آن مقدار بدست می‌آید. از الگوریتم‌های محبوب در زمینه جستجو می‌توان به الگوریتم ساده *جستجوی خطی* (*linear search*) و الگوریتم سریع‌تر اما پیچیده‌تر *جستجوی باینری* (*binary search*) اشاره کرد.

در مرتب‌سازی (sorting) مبادرت به قرار دادن داده‌ها به ترتیب صعودی یا نزولی، براساس یک یا چند کلید مرتب‌سازی (sort keys) می‌شود. برای مثال، اسامی موجود در یک دفتر تلفن را می‌توان براساس الفبا، حساب‌های بانکی را براساس شماره حساب بانکی، لیست حقوق کارمندان را براساس شماره تامین اجتماعی مرتب کرد. قبلاً در مورد مرتب‌سازی درجی (interstion) و مرتب‌سازی انتخابی (selection) در فصل‌های هفتم و هشتم توضیحاتی داده شده است. در این فصل به توضیح الگوریتم مرتب‌سازی کارآمدتری بنام *مرتب‌سازی ادغامی* (*merge sort*) خواهیم پرداخت.

جدول، شکل 20-1 حاوی الگوریتم‌های مرتب‌سازی و جستجوی مطرح شده در این کتاب است. همچنین در این فصل به معرفی نماد O بزرگ (Big O) می‌پردازیم، که برای برآورد کردن بدترین زمان اجرای یک الگوریتم است، یعنی حداکثر کاری که باید یک الگوریتم انجام دهد تا مسئله‌ای حل گردد.

20-2 الگوریتم‌های جستجو

جستجوی یک شماره تلفن، دسترسی به یک وب سایت و پیدا کردن معنی یک کلمه در واژه‌نامه همگی مستلزم جستجو در میان حجم زیادی از داده‌ها است. الگوریتم‌های جستجو همگی برای برآورده کردن چنین اهدافی بکار گرفته می‌شوند، یافتن عنصری که با کلید جستجو مطابقت دارد، اگر چنین عنصری وجود داشته باشد، پس وجود دارد. با این وجود، تمام این الگوریتم‌ها در برخی از موارد با یکدیگر تفاوت دارند. اصلی‌ترین تفاوت در میزان سعی است که برای کامل کردن جستجو صرف می‌کنند. یکی از

روش‌های توصیف این سعی و تلاش استفاده از نماد Big O می‌باشد. در الگوریتم‌های جستجو و مرتب‌سازی، این مقدار وابسته به تعداد عناصر داده است.

در فصل هفتم، در مورد الگوریتم جستجوی خطی بحث کرده‌ایم، که یک الگوریتم ساده بوده و پیاده‌سازی آن آسان می‌باشد. اکنون در ارتباط با میزان کارایی این الگوریتم که با نماد Big O اندازه‌گیری می‌شود، صحبت می‌کنیم. سپس، به معرفی یک الگوریتم جستجوی می‌پردازیم که نسبتاً از کارایی مناسبی برخوردار است، اما پیچیده بوده و پیاده‌سازی آن مشکل است.

20-2-1 کارایی جستجوی خطی

فرض کنید الگوریتم ساده‌ای داریم که تعیین می‌کند آیا عنصر اول در یک بردار معادل با عنصر دوم در بردار است یا خیر. اگر بردار 10 عنصر داشته باشد، این الگوریتم مستلزم یک مقایسه است. اگر بردار دارای 1000 عنصر باشد، هنوز هم الگوریتم مستلزم یک مقایسه است. در واقع، الگوریتم بطور کامل مستقل از تعداد عناصر در بردار عمل می‌کند.

فصل	الگوریتم
الگوریتم‌های جستجو	
هفتم	جستجوی خطی
بیستم	جستجوی باینری
بیست و یکم	درخت جستجوی باینری
الگوریتم مرتب‌سازی	
هفتم	مرتب‌سازی درجی
هشتم	مرتب‌سازی انتخابی
بیستم	مرتب‌سازی ادغامی
بیست و یکم	مرتب‌سازی درخت باینری

شکل 20-1 | الگوریتم‌های جستجو و مرتب‌سازی در این کتاب.

گفته می‌شود این الگوریتم دارای زمان اجرای ثابت (constant runtime) است که با نماد O(1) نشان داده می‌شود. الگوریتمی که دارای O(1) است، ضرورتاً مستلزم یک مقایسه نمی‌باشد. O(1) به این معنی است که تعداد مقایسه‌ها ثابت است، مقدار آن با افزایش سایز بردار رشد نمی‌کند. الگوریتمی که تست می‌کند آیا اولین عنصر از بردار برابر با هر سه عنصر بعدی است یا خیر همیشه مستلزم انجام سه مقایسه است، اما در نماد Big O با O(1) عرضه می‌شود. غالباً تلفظ O(1) بصورت "on the order of 1" یا ساده‌تر از آن "order 1" است.

الگوریتمی که تست می‌کند آیا اولین عنصر از بردار برابر با هر تعداد از عناصر دیگر در بردار است یا خیر، مستلزم انجام n-1 مقایسه است، که در این رابطه n تعداد عناصر موجود در بردار می‌باشد. اگر بردار دارای

10 عنصر باشد، این الگوریتم مستلزم انجام 9 مقایسه است. اگر بردار دارای 1000 عنصر باشد، این الگوریتم نیازمند 999 مقایسه خواهد بود. همانطوری که n بزرگتر می‌شود، بخش n از عبارت هم «مسلط» شده و تفریق از یک بی‌اهمیت می‌شود. به همین دلیل به الگوریتمی که مستلزم انجام n-1 مقایسه است (همانند موردی که بیان کردیم) گفته شود دارای O(n) است. به الگوریتمی با O(n) گفته می‌شود که دارای *زمان اجرای خطی* (*linear runtime*) است. تلفظ O(n) بصورت "on order of n" یا ساده‌تر "order n" است.

حل فرض کنید الگوریتمی داریم که تست می‌کند آیا عنصر از بردار در جای دیگری از بردار تکثیر شده است یا خیر. بایستی عنصر اول با هر عنصری در بردار مقایسه شود. عنصر دوم بایستی با هر عنصری بجز عنصر اول مقایسه شود (در بار اول مقایسه شده است). عنصر سوم بایستی با هر عنصری بجز دو عنصر اول مقایسه شود. در پایان، این الگوریتم با انجام $n^2/2 - n/2$ یا $1+2+\ldots+(n-2)+(n+1)$ مقایسه خاتمه می‌یابد. همانطوری که n افزایش می‌یابد، عبارت n^2 مسلط شده و عبارت n بی‌اهمیت می‌شود. مجدداً نماد Big O عبارت n^2 را متمایز کرده، $n^2/2$ را باقی می‌گذارد. همانطوری که بزودی ملاحظه خواهید کرد، فاکتورهای ثابت در نماد Big O در نظر گرفته نمی‌شود.

نماد Big O علاقمند به نحوه رشد زمان اجرای الگوریتم در ارتباط با تعداد ایتم‌های پردازش شده است. فرض کنید الگوریتمی مستلزم n^2 مقایسه است. با چهار عنصر، الگوریتم مستلزم انجام 16 مقایسه، با هشت عنصر مستلزم 64 مقایسه خواهد بود. با این الگوریتم، با دو برابر شدن عناصر، تعداد مقایسه‌ها چهار برابر می‌شود. به الگوریتمی توجه کنید که مستلزم $n^2/2$ مقایسه است. با چهار عنصر، الگوریتم نیازمند هشت مقایسه، با هشت عنصر نیازمند 32 مقایسه خواهد بود. مجدداً با دو برابر شدن تعداد عناصر، تعداد مقایسه‌ها چهار برابر می‌شود. هر دو این الگوریتم‌ها براساس مربع n افزایش می‌یابند، از اینرو Big O ثابت‌ها را در نظر نمی‌گیرد و هر دو الگوریتم با $O(n^2)$ نشان داده می‌شود که از آن بعنوان *زمان اجرای درجه دوم* (*quadratic runtime*) یاد می‌شود. تلفظ آن بصورت "on order of n-squard" یا ساده‌تر "order n-squard" است.

زمانیکه n کوچک باشد، الگوریتم‌های $O(n^2)$ تاثیر قابل ملاحظه‌ای درکارایی نخواهند داشت. اما زمانیکه n افزایش یابد، متوجه کاهش کارایی خواهید شد. یک الگوریتم $O(n^2)$ که بر روی برداری با یک میلیون عنصر کار می‌کند، می‌تواند نیازمند یک تریلیون عملیات باشد (که هر یک می‌تواند مستلزم اجرای چندین دستورالعمل ماشین باشد). انجام چنین کاری می‌تواند به چند ساعت نیاز داشته باشد. برداری با یک میلیون عنصر، نیازمند یک عملیاتبه تعداد، عدد 1 با 18 صفر بتوان 2 است، عددی بسیار بزرگی که الگوریتم بتواند با آن کار کند. نوشتن الگوریتم‌های $O(n^2)$ آسان است، همانطوری که در فصل‌های قبلی با آنها

مواجه شده‌اید. در این فصل شاهد الگوریتم‌های با واحدهای Big O مناسب‌تر خواهید بود. غالباً چنین الگوریتم‌های نیازمند کمی هوشیاری و دقت بیشتر در پیاده‌سازی هستند، اما کارایی که بدست می‌دهند بسیار ارزشمندتر است، بویژه زمانیکه n مقدار بزرگتری دریافت می‌کند و الگوریتم در برنامه‌های بزرگتر بکار گرفته می‌شود.

الگوریتم جستجوی خطی در زمان O(n) اجرا می‌شود. بدترین حالت در این الگوریتم زمانی است که باید تمام عناصر بررسی شوند تا مشخص گردد آیا کلید جستجو در بردار وجود دارد یا خیر. اگر سایز بردار دو برابر گردد، تعداد مقایسه‌های انجام گرفته نیز دو برابر می‌شود. توجه کنید که اگر عنصری که مطابق با کلید جستجو است نزدیک به ابتدای بردار قرار گرفته باشد، کارایی جستجوی خطی مناسب خواهد بود. اما بدنبال الگوریتمی هستیم که از میانگین کارایی مناسبی در تمام حالات جستجو برخوردار باشد، چه عنصر در ابتدا، میانه یا انتهای بردار قرار گرفته باشد.

پیاده‌سازی جستجوی خطی کار آسانی است، اما در مقایسه با سایر الگوریتم‌های جستجو، آهسته‌تر عمل می‌کند. اگر برنامه‌ای نیاز به انجام جستجو در میان بردارهای بزرگ داشته باشد، بهتر خواهد بود تا الگوریتم بهتر با کارایی بالاتر همانند جستجو باینری را بکار گرفت که در بخش بعدی به توضیح آن خواهیم پرداخت.

20-2-2 جستجوی باینری

الگوریتم *جستجوی باینری* (*binary search*) به نسبت الگوریتم جستجوی خطی از کارایی بالاتری برخوردار است، اما نیازمند آن است که ابتدا بردار مرتب گردد. اینکار فقط زمانی ارزش دارد که بردار فقط یکبار مرتب شده، و به دفعات زیاد جستجو می‌گردد یا زمانیکه برنامه جستجو کننده سخت بدنبال کارایی باشد. در اولین تکرار، این الگوریتم عنصر وسط در بردار را تست می‌کند. اگر این عنصر مطابق با کلید جستجو باشد، الگوریتم پایان می‌پذیرد. فرض کنید بردار به ترتیب صعودی مرتب شده باشد، پس اگر کلید جستجو کمتر از عنصر وسط باشد، کلید جستجو نمی‌تواند با هیچ یک از عناصر موجود در نیمه دوم بردار مطابقت داشته باشد و الگوریتم کار را فقط با نیمه اول بردار ادامه می‌دهد (یعنی از عنصر اول تا رسیدن به عنصر وسط، خود عنصر وسط در نظر گرفته نمی‌شود). اگر کلید جستجو بزرگتر از عنصر وسط باشد، پس کلید جستجو نمی‌تواند با هیچ یک از عناصر موجود در نیمه اول بردار مطابقت داشته باشد و الگوریتم کار را فقط با نیمه دوم بردار ادامه می‌دهد (یعنی پس از عنصر وسط تا عنصر آخر). در هر تکرار مقدار وسط از بخش باقیمانده بردار تست می‌شود. اگر عنصر مطابق با کلید جستجو نباشد، الگوریتم نصف باقیمانده از عناصر را در نظر نمی‌گیرد. الگوریتم یا با یافتن عنصر مورد نظر خاتمه می‌یابد یا اینکه عنصر یافت نشده و زیر بردار به سایز صفر می‌رسد. بعنوان یک مثال به بردار 15 عنصری زیر توجه کنید

```
2 3 5 10 27 30 34 51 56 65 77 81 82 93 99
```

فرض کنید کلید جستجو 65 باشد. برنامه با استفاده از الگوریتم جستجوی باینری شروع به جستجو می‌کند. ابتدا بررسی می‌کند که آیا 51 کلید جستجو است یا خیر (چرا که 51 عنصر وسط در بردار است). کلید جستجو (65) بزرگتر از 51 می‌باشد، از اینرو 51 به همراه نیمه اول بردار به کنار گذاشته می‌شوند (تمام عناصر کوچکتر از 51). سپس الگوریتم بررسی می‌کند که آیا 81 (عنصر وسط از بردار باقیمانده) با کلید جستجو مطابقت می‌کند یا خیر. کلید جستجو (65) کوچکتر از 81 است، از اینرو، 81 به همراه عناصر بزرگتر از 81 به کنار گذاشته می‌شوند. فقط پس از انجام دو تست، الگوریتم تعداد عناصری که باید بررسی شوند را به سه کاهش داده است (65, 56 و 77). سپس الگوریتم به بررسی 65 می‌پردازد (که به راستی با کلید جستجو مطابقت داد) و شاخص آن عنصر (9) را که حاوی 65 است برگشت می‌دهد. در این مورد خاص، این الگوریتم با سه عملیات مقایسه موفق به تعیین مطابقت عنصری با کلید جستجو شده است. در حالیکه جستجوی خطی برای انجام اینکار مستلزم انجام 10 مقایسه است.

در برنامه شکل‌های 2–20 و 3–20 کلاس **BinarySearch** و توابع عضو آن تعریف شده‌اند. این کلاس شبیه کلاس **LinearSearch** در فصل هفتم است. این کلاس دارای یک سازنده، یک تابع جستجو (**binarySearch**)، تابع **disyplayElements**، دو عضو داده خصوصی و یک تابع کمکی بنام **displaySubElements** است. در خطوط 28-18 از شکل 3–20 یک سازنده تعریف شده است. پس از مقداردهی بردار با مقادیر تصادفی صحیح از 99-10 در خطوط 25-24، خط 27 تابع استاندارد کتابخانه‌ای **sort** را بر روی بردار **data** فراخوانی می‌کند. تابع **sort** دو تکرار شونده با دسترسی تصادفی دریافت و عناصر موجود در بردار را به ترتیب صعودی مرتب می‌کند. این نوع تکرار شونده، تکرار شونده‌ای است که دارای مجوز دسترسی به هر ایتم موجود در بردار را در هر زمان دارد. در این برنامه از **data.being()** و **data.end()** برای احاطه کردن کل بردار استفاده کرده‌ایم. بخاطر داشته باشد که الگوریتم جستجوی باینری فقط بر روی بردار مرتب شده کار می‌کند.

خطوط 36-31 تعریف کننده تابع **binarySearch** هستند. کلید جستجو به پارامتر **searchElement** ارسال می‌شود (خط 31). خطوط 35-33 مبادرت به محاسبه شاخص پایین **low**، شاخص بالا **high** و شاخص وسط یا میانی **middle** در بخشی از بردار می‌کنند که برنامه در حال حاضر آنرا جستجو می‌کند. در ابتدای کار تابع، **low** برابر صفر، **high** برابر سایز بردار منهای **1** و **middle** برابر میانگین این دو مقدار است. خط 36 مبادرت به مقداردهی اولیه **location** با **1-** می‌کند، مقداری که در صورت عدم دریافت کلید جستجو، برگشت داده خواهد شد. حلقه موجود در خطوط 58-38 تا زمانیکه **low** بزرگتر از **high** شده یا **location** برابر **1-** نشود ادامه می‌یابد.

```
1       // Fig 20.2: BinarySearch.h
2       // Class that contains a vector of random integers and a function
3       // that uses binary search to find an integer.
```

```
4       #include <vector>
5       using std::vector;
6
7       class BinarySearch
8       {
9       public:
10         BinarySearch( int ); // constructor initializes vector
11         int binarySearch( int ) const;// perform a binary search on vector
12         void displayElements() const; // display vector elements
13      private:
14         int size; // vector size
15         vector< int > data; // vector of ints
16         void displaySubElements(int,int) const;// display range of values
17      }; // end class BinarySearch
```

شکل 2–20 | تعریف کلاس BinarySearch.

```
1       // Fig 20.3: BinarySearch.cpp
2       // BinarySearch class member-function definition.
3       #include <iostream>
4       using std::cout;
5       using std::endl;
6
7       #include <cstdlib> // prototypes for functions srand and rand
8       using std::rand;
9       using std::srand;
10
11      #include <ctime> // prototype for function time
12      using std::time;
13
14      #include <algorithm> // prototype for sort function
15      #include "BinarySearch.h" // class BinarySearch definition
16
17      //constructor initializes vector with random ints and sorts the vector
18      BinarySearch::BinarySearch( int vectorSize )
19      {
20         size = (vectorSize > 0 ? vectorSize : 10); // validate vectorSize
21         srand( time( 0 ) ); // seed using current time
22
23         // fill vector with random ints in range 10-99
24         for ( int i = 0; i < size; i++ )
25            data.push_back( 10 + rand() % 90 ); // 10-99
26
27         std::sort( data.begin(), data.end() ); // sort the data
28      } // end BinarySearch constructor
29
30      // perform a binary search on the data
31      int BinarySearch::binarySearch( int searchElement ) const
32      {
33         int low = 0; // low end of the search area
34         int high = size - 1; // high end of the search area
35         int middle = ( low + high + 1 ) / 2; // middle element
36         int location = -1; // return value; -1 if not found
37
38         do // loop to search for element
39         {
40            // print remaining elements of vector to be searched
41            displaySubElements( low, high );
42
43            // output spaces for alignment
44            for ( int i = 0; i < middle; i++ )
45              cout << "   ";
46
```

```
47            cout << " * " << endl; // indicate current middle
48
49            // if the element is found at the middle
50            if ( searchElement == data[ middle ] )
51               location = middle; // location is the current middle
52            else if (searchElement < data[ middle ]) // middle is too high
53               high = middle - 1; // eliminate the higher half
54            else // middle element is too low
55               low = middle + 1; // eliminate the lower half
56
57            middle = ( low + high + 1 ) / 2; // recalculate the middle
58         } while ( ( low <= high ) && ( location == -1 ) );
59
60         return location; // return location of search key
61      } // end function binarySearch
62
63      // display values in vector
64      void BinarySearch::displayElements() const
65      {
66         displaySubElements( 0, size - 1 );
67      } // end function displayElements
68
69      // display certain values in vector
70      void BinarySearch::displaySubElements( int low, int high ) const
71      {
72         for ( int i = 0; i < low; i++ ) // output spaces for alignment
73            cout << "   ";
74
75         for (int i = low; i <= high; i++)//output elements left in vector
76            cout << data[ i ] << " ";
77
78         cout << endl;
79      } // end function displaySubElements
```

شکل 3-20 | تعریف تابع عضو کلاس BinarySearch.

خط 50 تست می‌کند که آیا مقدار موجود در عنصر **middle** برابر با **searchElement** است یا خیر. اگر چنین باشد، خط 51 مبادرت به تخصیص **middle** به **location** می‌کند، سپس حلقه خاتمه یافته و **location** به فراخوان برگردانده می‌شود. هر تکرار حلقه مبادرت به تست یک مقدار می‌کند (خط 50) و نصف باقیمانده مقادیر در بردار را به کنار می‌گذارد (خط 53 یا 55).

حلقه خطوط 41-25 از شکل 4-20 تا زمانیکه کاربر مقدار 1- وارد کند، ادامه می‌یابد. برای هر عدد دیگری که کاربر وارد کند، برنامه یک جستجوی باینری بر روی داده انجام می‌دهد و تعیین می‌کند که آیا با عنصری در بردار مطابقت می‌کند یا خیر.

کارایی جستجوی باینری

در بدترین وضعیت، جستجوی یک بردار مرتب شده با 1023 عنصر با استفاده از الگوریتم جستجوی باینری فقط 10 مقایسه لازم دارد. با تقسیم متوالی 1023 به 2 و گرد کردن آن مقادیر 1,3,7,15,31,63,127,255,511 و صفر بدست می‌آیند. عدد $(2^{10}-1)$ 1023 به 2 فقط 10 بار تقسیم می‌شود تا مقدار صفر بدست آید، که مشخص می‌کند که هیچ عنصری برای تست باقی نمانده است.

تقسیم بر 2 معادل با یک مقایسه در الگوریتم جستجوی باینری است. از اینرو، برداری با 2^{20} - 1,048,575 (1 عنصر حداکثر به 20 مقایسه برای یافتن کلید نیاز دارد و برداری با بیش از یک تریلیون حداکثر به 30 مقایسه نیاز خواهد داشت، که در مقایسه با جستجوی خطی، بهبود بسیار عظیمی دیده می‌شود. اگر این تعداد به الگوریتم جستجوی خطی داده شود، بطور میانگین نیاز به 500 میلیون مقایسه خواهد بود. حداکثر تعداد مقایسه مورد نیاز در جستجوی باینری بر روی هر بردار مرتب شده، توانی از 2 است بزرگتر از تعداد عناصر در بردار که بصورت خ مشابه صورت تقریباً با نر هاشود. رشد تمام لگاریتم‌نشان داده می $\log_2^n$ برای O(log n)توان پایه را در نظر نگرفت. نتیجه اینکار بصورت می Big Oگیرد، از اینرو در نماد می شود و شناخته می)*logarithmic runtime*(*زمان اجرای لگاریتمی*جستجوی باینری خواهد بود، که بعنوان ی‌اهم‌تیروگلا 02–3□است. "order *n* log "تر یا ساده "on the order of log *n*"تلفظ آن بصورت

ار ی‌دربراک ی‌اهه‌مانرب ی‌اهش‌خب‌ن یرتمهم زا ی‌کی‌اهه‌داد ی‌زاسب‌ترم□سازی‌مرتب

توان شوند، از اینروست که می‌ها توسط شماره حساب مرتب می‌دهد. در یک بانک تمام چک‌تشکیل می های خود را براساس نام‌ت‌های تلفن لیسای در پایان هر ماه آماده کرد. شرکت‌صورتحساب جداگانه های تلفن آسانتر شود. در واقع هر سازمانی که با کنند تا یافتن شماره‌خانوادگی و همچنین نام مرتب می طابترا رد ه‌طقن‌یرتمهم□های خود است.سازی داده‌حجم زیادی از اطلاعات سر و کار دارد نیازمند مرتب و مهم نیست که کدام الگوریتم باشدمی‌دار مرتب شده که یک بر ،سازی در نتیجه پایانی است‌با مرتب برای مرتب کردن بردار بکار گرفته شده است. الگوریتم انتخاب شده فقط بر روی زمان اجرا و حافظه سازی انتخابی و درجی پرداختیم که های قبلی، به معرفی مرتب‌مورد استفاده برنامه تاثیر دارد. در فصل اما از کارایی ضعیفی برخوردار بودند. در بخش بعدی به بررسی کارایی این دو سازی آسان پیاده‌دارای پردازیم می (merge)پردازیم. در پایان به بررسی الگوریتم ادغامی می Big Oالگوریتم با استفاده از نماد ppc.40_02giF :4.02 giF //□ hcraeSyraniB // سازی آن مشکل است.که سریعتر بوده اما پیاده

test program

3–20 الگوریتم‌های مرتب‌سازی

مرتب‌سازی داده‌ها یکی از مهمترین بخش‌های برنامه‌های کاربردی را تشکیل می‌دهد. در یک بانک تمام چک‌ها توسط شماره حساب مرتب می‌شوند، از اینروست که می‌توان صورتحساب جداگانه‌ای در پایان هر ماه آماده کرد. شرکت‌های تلفن لیست‌های خود را براساس نام خانوادگی و همچنین نام مرتب می‌کنند تا یافتن شماره‌های تلفن آسانتر شود. در واقع هر سازمانی که با حجم زیادی از اطلاعات سر و کار دارد نیازمند مرتب‌سازی داده‌های خود است.

مهمترین نقطه در ارتباط با مرتب‌سازی در نتیجه پایانی است، که یک بردار مرتب شده می‌باشد و مهم نیست که کدام الگوریتم برای مرتب کردن بردار بکار گرفته شده است. الگوریتم انتخاب شده فقط بر روی زمان اجرا و حافظه مورد استفاده برنامه تاثیر دارد. در فصل‌های قبلی، به معرفی مرتب‌سازی انتخابی و درجی پرداختیم که دارای پیاده‌سازی آسان اما از کارایی ضعیفی برخوردار بودند. در بخش بعدی به بررسی کارایی این دو الگوریتم با استفاده از نماد Big O می‌پردازیم. در پایان به بررسی الگوریتم ادغامی (merge) می‌پردازیم که سریعتر بوده اما پیاده‌سازی آن مشکل است.

```
1       // Fig 20.4: Fig20_04.cpp
2       // BinarySearch test program.
3       #include <iostream>
4       using std::cin;
5       using std::cout;
6       using std::endl;
7
8       #include "BinarySearch.h" // class BinarySearch definition
9
10      int main()
11      {
12         int searchInt; // search key
13         int position; // location of search key in vector
14
15         // create vector and output it
16         BinarySearch searchVector ( 15 );
17         searchVector.displayElements();
18
19         // get input from user
20         cout << "\nPlease enter an integer value (-1 to quit): ";
21         cin >> searchInt; // read an int from user
22         cout << endl;
23
24         // repeatedly input an integer; -1 terminates the program
25         while ( searchInt != -1 )
26         {
27            // use binary search to try to find integer
28            position = searchVector.binarySearch( searchInt );
29
30            // return value of -1 indicates integer was not found
31            if ( position == -1 )
32               cout << "The integer " << searchInt << " was not found.\n";
33            else
34               cout << "The integer " << searchInt
35                  << " was found in position " << position << ".\n";
36
37            // get input from user
38            cout << "\n\nPlease enter an integer value (-1 to quit): ";
39            cin >> searchInt; // read an int from user
40            cout << endl;
41         } // end while
42
43         return 0;
44      } // end main
```

```
 26 31 33 38 47 49 49 67 73 74 82 89 90 91 95

Please enter an integer value (-1 to quit): 38
26 31 33 38 47 49 49 67 73 74 82 89 90 91 95
                     *
```

```
26 31 33 38 47 49 49
         *
26 31 33 38 47 49 49 67 73 74 82 89 90 91 95

Please enter an integer value (-1 to quit): 38
26 31 33 38 47 49 49 67 73 74 82 89 90 91 95
                     *
26 31 33 38 47 49 49
         *
The integer 38 was found in position 3.

Please enter an integer value (-1 to quit): 91
26 31 33 38 47 49 49 67 73 74 82 89 90 91 95
                     *
                       73 74 82 89 90 91 95
                                  *
                                    90 91 95
                                       *
The integer 91 was found in position 13.

Please enter an integer value (-1 to quit): 25
26 31 33 38 47 49 49 67 73 74 82 89 90 91 95
                     *
26 31 33 38 47 49 49
         *
26 31 33
   *
26
 *
The integer 25 was not found.

Please enter an integer value (-1 to quit): -1
```

شکل 4-20 | برنامه تست BinarySearch.

20-3-1 کارایی مرتب‌سازی انتخابی

پیاده‌سازی الگوریتم *مرتب‌سازی انتخابی* (*selection sort*) کار آسانی است اما از کارایی پایینی برخوردار است. در اولین حرکت این الگوریتم، کوچکترین عنصر در بردار انتخاب شده و مکان آن با اولین عنصر عوض می‌شود. در دومین حرکت، دومین عنصر کوچک (کوچکترین عنصر در میان عناصر باقیمانده) انتخاب و مکان آن با دومین عنصر عوض می‌شود. الگوریتم تا آخرین حرکت که انتخاب دومین عنصر بزرگ و عوض کردن آن با دومین عنصر آخر ادامه می‌دهد، و بزرگترین عنصر در آخرین شاخص گذاشته می‌شود. پس از *i*th تکرار، کوچکترین عناصر i بردار بصورت صعودی مرتب می‌شوند.

الگوریتم مرتب‌سازی انتخابی n-1 بار تکرار می‌شود و در هر بار جابجایی کوچکترین عنصر باقی مانده در موقعیت مرتب شده خود جای می‌گیرد. پیدا کردن کوچکترین عنصر باقی مانده مستلزم n-1 مقایسه در اولین تکرار، n-2 در زمان دومین تکرار، سپس n-3,…,3,2,1 تکرار است. در نتیجه مجموع n2 - n)/2) یا n(n - 1)/2 مقایسه بدست می‌آید. در نماد Big O، کوچکترین حالت و ثابت‌ها به کنار گذاشته می‌شوند و در نتیجه $O(n^2)$ بدست می‌آید.

20-3-2 کارایی مرتب‌سازی درجی

مرتب‌سازی درجی (*insertin sort*) از پیاده‌سازی ساده‌ای برخوردار بوده، اما از کارایی چندانی برخوردار نیست. در اولین تکرار این الگوریتم، دومین عنصر در بردار گرفته شده و اگر کوچکتر از اولین عنصر باشد، آنرا با عنصر اول عوض می‌کند. در دومین تکرار به سومین عنصر نگاه شده و آنرا در موقعیت صحیح با نگاهی به دو عنصر اول درج می‌کند، از اینرو هر سه عنصر مرتب می‌شوند، در *i*th تکرار این الگوریتم، اولین عناصر i در بردار اصلی مرتب شده خواهند بود.

مرتب‌سازی درجی، n-1 بار تکرار می‌شود، یک عنصر در موقعیت مناسب در میان عناصر مرتب شده تا بدین جا، قرار داده می‌شود. در هر تکرار، تعیین می‌شود که عنصر درج شونده نیاز به مقایسه با هر عنصر قبل از خود در بردار دارد یا خیر. در بدترین حالت، نیازمند n-1 مقایسه است. هر عبارت تکرار در زمان O(n) اجرا می‌شود. در تعیین نماد Big O، عبارات تودرتو باید تعداد مقایسه‌ها به هم ضرب شوند. برای هر تکرار حلقه خارجی، تعداد مشخصی از تکرار حلقه داخلی وجود دارد. در این الگوریتم، برای هر O(n) تکرار حلقه خارجی، O(n) تکرار برای حلقه داخلی وجود دارد که نتیجه آن O(n*n) یا $O(n^2)$ است.

20-3-3 مرتب‌سازی ادغامی (پیاده‌سازی بازگشتی)

مرتب‌سازی ادغامی از الگوریتم‌های مرتب‌سازی با کارایی بالا بوده اما به لحاظ مفهومی بسیار پیچیده‌تر از مرتب‌سازی انتخابی و درجی می‌باشد. الگوریتم مرتب‌سازی ادغامی اقدام به مرتب کردن یک بردار با تقسیم آن به دو زیر بردار با سایز برابر کرده، هر زیر بردار را مرتب و سپس دو زیر بردار را با هم ادغام می‌کند تا بردار بزرگتر بدست آید. در یک بردار با تعداد عناصر فرد، الگوریتم دو زیر بردار ایجاد می‌کند که یکی بیش از دیگری یک عنصر بیشتر دارد.

پیاده‌سازی مرتب‌سازی ادغامی در این بخش بصورت بازگشتی است. حالت پایه یا مبنا، برداری است با یک عنصر. البته، بردار یک عنصری، مرتب شده است، از اینرو مرتب‌سازی ادغامی بلافاصله به هنگام برخورد با بردار یک عنصری، برگشت داده می‌شود. گام بازگشتی مبادرت به تقسیم برداری با دو یا چند عنصر به دو زیر بردار با سایز برابر می‌کند، بطور بازگشتی هر زیر بردار مرتب شده، سپس آن دو زیر بردار به هم می‌پیوندند.

فرض کنید که در حال حاضر الگوریتم دارای بردارهای کوچکتر مرتب شده A و B است که می‌خواهند به با ادغام شوند تا یک بردار بزرگ مرتب شده بوجود آید، بردار A:

```
4 10 34 56 77
```

و بردار B:

```
5 30 51 52 93
```

کوچکترین عنصر در A عدد 4 است (در شاخص صفر). کوچکترین عنصر در B عدد 5 است (در شاخص صفر از B). برای تعیین کوچکترین عنصر در بردار بزرگ، الگوریتم مبادرت به مقایسه 4 و 5 می‌کند. مقدار موجود در A کوچکتر بوده، از اینرو 4 تبدیل به اولین عنصر در بردار ادغام شده می‌شود. الگوریتم با مقایسه 10 (دومین عنصر در A) با 5 (اولین عنصر در B) تعیین می‌کند که مقدار موجود در B کوچکتر است، از اینرو 5 تبدیل به دومین عنصر در بردار بزرگ می‌شود. الگوریتم به مقایسه 10 با 30، ادامه داده و 10 تبدیل به سومین عنصر در بردار می‌شود و اینکار تا پایان ادامه می‌یابد.

برنامه شکل 5–20 کلاس **MergeSort** را تعریف کرده و خطوط 34-31 از شکل 6–20 تابع **sort** را تعریف کرده‌اند. خط 33 تابع **sortSubVector** با آرگومان‌های صفر و size-1 فراخوانی کرده است. این آرگومان‌ها متناظر با شاخص‌های ابتدا و انتهای برداری هستند که مرتب شده‌اند و سبب می‌شوند تا **sortSubVector** بر روی کل بردار عمل کند. تابع **sortSubVector** در خطوط 61-37 تعریف شده است. خط 40 تست کننده حالت پایه است. اگر سایز بردار 1 باشد، تابع بردار را به دو قسمت تقسیم کرده، بطور بازگشتی تابع **sortSubVector** را برای مرتب کردن دو زیر بردار فراخوانی کرده، سپس آنها را با هم ادغام می‌کند. خط 55 بصورت بازگشتی تابع **sortSubVetor** را بر روی نیمه اول بردار فراخوانی کرده، و خط 56 بصورت بازگشتی تابع **sortSubVector** را بر روی نیمه دوم بردار فراخوانی می‌کند. زمانیکه این دو تابع باز گردند، هر نیمه از بردار مرتب شده خواهد بود.

```
1       // Fig 20.05: MergeSort.h
2       // Class that creates a vector filled with random integers.
3       // Provides a function to sort the vector with merge sort.
4       #include <vector>
5       using std::vector;
6
7       // MergeSort class definition
8       class MergeSort
9       {
10      public:
11         MergeSort( int ); // constructor initializes vector
12         void sort(); // sort vector using merge sort
13         void displayElements() const; // display vector elements
14      private:
15         int size; // vector size
16         vector< int > data; // vector of ints
17         void sortSubVector( int, int ); // sort subvector
18         void merge( int, int, int, int ); // merge two sorted vectors
19         void displaySubVector( int, int ) const; // display subvector
20      }; // end class SelectionSort
```

شکل 5–20 | تعریف کلاس MergSort.

```
1       // Fig 20.06: MergeSort.cpp
2       // Class MergeSort member-function definition.
3       #include <iostream>
4       using std::cout;
5       using std::endl;
6
7       #include <vector>
```

```
8       using std::vector;
9
10      #include <cstdlib> // prototypes for functions srand and rand
11      using std::rand;
12      using std::srand;
13
14      #include <ctime> // prototype for function time
15      using std::time;
16
17      #include "MergeSort.h" // class MergeSort definition
18
19      // constructor fill vector with random integers
20      MergeSort::MergeSort( int vectorSize )
21      {
22         size = (vectorSize > 0 ? vectorSize : 10);// validate vectorSize
23         srand(time(0));// seed random number generator using current time
24
25         // fill vector with random ints in range 10-99
26         for ( int i = 0; i < size; i++ )
27            data.push_back( 10 + rand() % 90 );
28      } // end MergeSort constructor
29
30      // split vector, sort subvectors and merge subvectors into sorted vector
31      void MergeSort::sort()
32      {
33         sortSubVector( 0, size - 1 ); // recursively sort entire vector
34      } // end function sort
35
36      // recursive function to sort subvectors
37      void MergeSort::sortSubVector( int low, int high )
38      {
39         // test base case; size of vector equals 1
40         if ( ( high - low ) >= 1 ) // if not base case
41         {
42            int middle1 = ( low + high ) / 2; //calculate middle of vector
43            int middle2 = middle1 + 1; // calculate next element over
44
45            // output split step
46            cout << "split:   ";
47            displaySubVector( low, high );
48            cout << endl << "          ";
49            displaySubVector( low, middle1 );
50            cout << endl << "          ";
51            displaySubVector( middle2, high );
52            cout << endl << endl;
53
54            // split vector in half; sort each half (recursive calls)
55            sortSubVector( low, middle1 ); // first half of vector
56            sortSubVector( middle2, high ); // second half of vector
57
58            // merge two sorted vectors after split calls return
59            merge( low, middle1, middle2, high );
60         } // end if
61      } // end function sortSubVector
62
63      // merge two sorted subvectors into one sorted subvector
64      void MergeSort::merge(int left, int middle1, int middle2, int right)
65      {
66         int leftIndex = left; // index into left subvector
67         int rightIndex = middle2; // index into right subvector
68         int combinedIndex = left; // index into temporary working vector
69         vector< int > combined( size ); // working vector
```

```
70
71          // output two subvectors before merging
72          cout << "merge:   ";
73          displaySubVector( left, middle1 );
74          cout << endl << "         ";
75          displaySubVector( middle2, right );
76          cout << endl;
77
78          // merge vectors until reaching end of either
79          while ( leftIndex <= middle1 && rightIndex <= right )
80          {
81             // place smaller of two current elements into result
82             // and move to next space in vector
83             if ( data[ leftIndex ] <= data[ rightIndex ] )
84                combined[ combinedIndex++ ] = data[ leftIndex++ ];
85             else
86                combined[ combinedIndex++ ] = data[ rightIndex++ ];
87          } // end while
88
89          if ( leftIndex == middle2 ) // if at end of left vector
90          {
91             while ( rightIndex <= right ) // copy in rest of right vector
92                combined[ combinedIndex++ ] = data[ rightIndex++ ];
93          } // end if
94          else // at end of right vector
95          {
96             while ( leftIndex <= middle1 ) // copy in rest of left vector
97                combined[ combinedIndex++ ] = data[ leftIndex++ ];
98          } // end else
99
100         // copy values back into original vector
101         for ( int i = left; i <= right; i++ )
102            data[ i ] = combined[ i ];
103
104         // output merged vector
105         cout << "         ";
106         displaySubVector( left, right );
107         cout << endl << endl;
108      } // end function merge
109
110      // display elements in vector
111      void MergeSort::displayElements() const
112      {
113         displaySubVector( 0, size - 1 );
114      } // end function displayElements
115
116      // display certain values in vector
117      void MergeSort::displaySubVector( int low, int high ) const
118      {
119         // output spaces for alignment
120         for ( int i = 0; i < low; i++ )
121            cout << "   ";
122
123         // output elements left in vector
124         for ( int i = low; i <= high; i++ )
125            cout << " " << data[ i ];
126      } // end function displaySubVector
```

شکل 6–20 | تعریف تابع عضو کلاس MergSort.

خط 59 تابع **merge** (خطوط 108-64) را بر روی بردار دو نیم شده فراخوانی می‌کند تا دو بردار مرتب شده با هم ترکیب و یک بردار مرتب شده بزرگتر بوجود آید.

خطوط 87-79 در تابع **merge** تا زمانیکه برنامه به انتهای زیر بردارها برسد، تکرار می‌شوند (حلقه). خط 83 تست می‌کند که کدام عنصر در ابتدای بردارها کوچکتر است. اگر عنصر موجود در بردار چپ کوچکتر باشد، خط 84 آنرا در موقعیت بردار ترکیبی قرار می‌دهد. اگر عنصر موجود در بردار راست کوچکتر باشد، خط 86 آنرا در موقعیت بردار ترکیبی قرار می‌دهد. زمانیکه حلقه **while** کامل شد (خط 87)، کل یک زیر بردار در بردار ترکیبی جای خواهد داشت، اما زیر بردار دیگر هنوز دارای داده است. خط 89 تست می‌کند که آیا بردار چپ به انتها رسیده است یا خیر. اگر چنین باشد، خطوط 92-91 بردار ترکیبی را با عناصر بردار راست پر می‌کنند. اگر بردار چپ به انتها نرسیده باشد، پس بایستی بردار راست به انتها رسیده باشد و خطوط 97-96 بردار ترکیبی را با عناصر بردار چپ پر می‌کنند. سرانجام، خطوط 101- 102 بردار ترکیبی را بر روی بردار اولیه کپی می‌کنند. برنامه شکل 7–20 یک شی **MergeSorg** ایجاد کرده و از آن استفاده می‌کند.

```
1       // Fig 20.07: Fig20_07.cpp
2       // MergeSort test program.
3       #include <iostream>
4       using std::cout;
5       using std::endl;
6
7       #include "MergeSort.h" // class MergeSort definition
8
9       int main()
10      {
11         // create object to perform merge sort
12         MergeSort sortVector( 10 );
13
14         cout << "Unsorted vector:" << endl;
15         sortVector.displayElements(); // print unsorted vector
16         cout << endl << endl;
17
18         sortVector.sort(); // sort vector
19
20         cout << "Sorted vector:" << endl;
21         sortVector.displayElements(); // print sorted vector
22         cout << endl;
23         return 0;
24      } // end main
```

```
Unsorted vector:
30 47 22 67 79 18 60 78 26 54

split:   30 47 22 67 79 18 60 78 26 54
         30 47 22 67 79
                        18 60 78 26 54

split:   30 47 22 67 79
         30 47 22
                  67 79
```

```
split:   30 47 22
         30 47
               22

split:   30 47
         30
            47

merge:   30
            47
         30 47

merge:   30 47
               22
         22 30 47
split:              67 79
                    67
                       79

merge:              67
                       79
                    67 79

merge:   22 30 47
                    67 79
         22 30 47 67 79

split:                    18 60 78 26 54
                          18 60 78
                                   26 54

split:                    18 60 78
                          18 60
                                78

split:                    18 60
                          18
                             60

merge:                    18
                             60
                          18 60

merge:                    18 60
                                78
                          18 60 78

split                               26 54
                                    26
                                       54

merge:                              26
                                       54
                                    26 54

merge:                    18 60 78
                                    26 54
                          18 26 54 60 78
merge:  22 30 47 67 79
                          18 26 54 60 78
        18 22 26 30 47 54 60 67 78 79
```

```
Sorted vector:
18 22 26 30 47 54 60 67 78 79
```

شکل 7–20 | برنامه تست MergeSort.

کارایی مرتب‌سازی ادغامی

کارایی مرتب‌سازی ادغامی در مقایسه با مرتب‌سازی‌های انتخابی و درجی بسیار بیشتر است. به اولین فراخوانی (غیر بازگشتی) تابع **sortSubVector** توجه کنید. نتیجه این فراخوانی، دو فراخوانی بازگشتی برای تابع **sortSubVector** با دو زیر بردار است که هر یک تقریباً نصف سایز بردار اولیه، سایز دارند و یک فراخوانی برای تابع **merge** صورت می‌گیرد. فراخوانی تابع **merge** در بدترین حالت، نیازمند n-1 مقایسه برای پر کردن بردار اولیه (اصلی) است که برابر O(n) است. نتیجه دو فراخوانی برای تابع **sortSubVector** فراخوانی بازگشتی بیش از چهار بار تابع **sortSubVector** است که هر یک با یک زیر بردار تقریباً یک چهارم سایز بردار اولیه، به همراه دو فراخوانی تابع **merge** است. فراخوانی این دو تابع **merge** هر یک مستلزم n/2 - 1 مقایسه است که برای کل تعداد مقایسه O(n) را داریم. این فرآیند ادامه می‌یابد تا هر فراخوانی **sortSubVector** دو فراخوانی دیگر **sortSubVector** و یک فراخوانی **merge** انجام دهد تا اینکه الگوریتم به برداری با یک عنصر برسد. در هر سطح، O(n) مقایسه برای ادغام زیر بردارها لازم است. هر سطح مبادرت به تقسیم سایز بردارها به نصف کرده، از اینرو دو برابر کردن سایز بردار نیازمند یک سطح بیشتر است. چهار برابر کردن سایز بردار نیازمند دو سطح بیشتر است. این الگو حالت لگاریتمی داشته و نتیجه آن $\log_2^n$ سطح است. این نتایج بصورت کلی نشاندهنده $O(n \log n)$ می‌باشند. جدول شکل 8–20 بطور خلاصه برخی از نتایج Big O از الگوریتم‌های جستجو و مرتب‌سازی را نشان می‌دهد.

الگوریتم	Big O	الگوریتم	Big O
الگوریتم‌های جستجو		***الگوریتم‌های مرتب‌سازی***	
جستجوی خطی	$O(n)$	مرتب‌سازی درجی	$O(n^2)$
جستجوی باینری	$O(\log n)$	مرتب‌سازی انتخابی	$O(n^2)$
جستجوی خطی به روش بازگشتی	$O(n)$	مرتب‌سازی ادغامی	$O(n \log n)$
جستجوی باینری به روش بازگشتی	$O(\log n)$	مرتب‌سازی حبابی	$O(n^2)$
		مرتب‌سازی سریع (quick sort)	$O(n^2)$ *در بدترین حالت*
			$O(n \log n)$ *حالت میانگین*

شکل 8–20 | الگوریتم‌های جستجو مرتب‌سازی با مقادیر Big O.

فصل بیست و یکم

ساختمان‌های داده

اهداف

- **فرم‌دهی ساختمان‌های دادهٔ متصل به هم با استفاده از مراجعه‌ها، کلاس‌های خود ارجاعی و بازگشتی.**
- **ایجاد و کار با ساختمان‌های داده دینامیکی، همانند لینک لیست‌ها، صف‌ها، پشته‌ها و درخت‌های باینری.**
- **آشنایی با نحوه عملکرد برنامه‌های ساختمان داده.**
- **آشنایی با نحوه ایجاد ساختمان‌های دادهٔ با قابلیت استفاده مجدد با بکارگیری کلاس‌ها، توارث و ترکیب.**

رئوس مطالب

21-1 مقدمه

در مورد ساختمان‌های داده با سایز ثابت همانند آرایه‌های یک بعدی و آرایه‌های دو بعدی مطالبی بیان کردیم. در این فصل در مورد ساختمان‌های داده دینامیکی که در زمان اجرا تغییر سایز می‌دهند صحبت خواهیم کرد. لیست‌های پیوندی (link list) مجموعه‌ای از عناصر داده هستند که عمل درج و حذف در هر جای لیست پیوندی ممکن است. پشته‌ها (stack) جزء ساختارهای مهم در کامپایلرها و سیستم‌های عامل هستند که عمل درج و حذف فقط از بالای (ابتدای) آن ممکن است. صف‌ها (queue) به مانند صف‌های انتظار می‌باشند. عمل درج یا اضافه کردن به صف از انتهای آن (tail) و عمل حذف از ابتدای (head) آن صورت می‌گیرد. درخت‌های باینری (binary trees)، جستجوی بسیار سریع، ذخیره‌سازی داده‌ها و کامپایل عبارات به زبان ماشین را تسهیل می‌کنند. البته این ساختمان‌های داده در کاربردهای متفاوت دیگری نیز به کار گرفته می‌شوند.

در این فصل در مورد هر کدام یک از این ساختمان‌های داده مطالبی بیان خواهیم و برنامه‌های برای ایجاد و نگهداری این ساختمان داده‌ها خواهیم نوشت. از کلاس‌ها، توارث و ترکیب برای ایجاد بسته‌ها (package) و ساختمان‌های داده برای افزایش قابلیت برنامه‌ها استفاده خواهیم کرد.

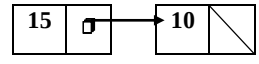
21-2 کلاس‌های خود ارجاعی

یک کلاس خود ارجاعی، حاوی بخشی است که اشارهٔ به یک شی کلاس، از همان نوع کلاس دارد. برای مثال، کلاس تعریف شده در زیر

```
class Node
{
public:
   Node( int ); // constructor
   void setData( int ); // set data member
   int getData() const; // get data member
 void setNextPtr( Node * ); // set pointer to next Node
 Node *getNextPtr() const; // get pointer to next Node
private:
   int data; // data stored in this Node
   Node *nextPtr; // pointer to another object of same type
}; // end ListNode
```

تعریف کننده نوع **Node** است. این نوع دارای دو عضو داده **private** به نام‌های **data** از نوع عدد صحیح و **nextPtr** است که به **Node** اشاره دارد. عضو **nextPtr** اشاره به شی از نوع **Node** دارد، همان نوع بعنوان کلاس جاری به عضو **nextPtr** بعنوان یک *لینک* (پیوند) نگاه می‌شود (به این معنی که **nextPtr** می‌تواند برای گره زدن یک شی از نوع **Node** به شی دیگری از همان نوع بکار گرفته شود). همچنین نوع **Node** دارای پنج تابع عضو است: یک سازنده که یک مقدار صحیح برای مقدار دهی اولیه عضو **data** دریافت می‌کند، یک تابع **setData** برای تنظیم مقدار عضو **data**، یک تابع **getData** برای برگشت دادن مقدار از عضو **data**، تابع **setNextPtr** بزای تنظیم مقدار عضو **nextPtr** و تابع **getNextPtr** برای برگشت دادن مقدار از عضو **nextPtr**.

شی‌های خود ارجاعی می‌توانند به یکدیگر متصل شده و ساختمان‌های داده مناسب‌تری همانند لیست‌ها، صف‌ها، پشته‌ها و درخت‌ها ایجاد کنند. شکل 1-21 تصویر دو شی خود ارجاعی متصل به هم را که تشکیل یک لیست را داده نشان می‌دهد. یک کاراکتر \ که در دومین شی خود ارجاعی جای گرفته نشان می‌دهد که لینک به شی دیگری اشاره نمی‌کند. معمولاً از **null** برای تعریف انتهای یک ساختمان داده استفاده می‌شود.



شکل 1-21 | شی‌های کلاس خود ارجاعی که به هم متصل شده‌اند.

خطای برنامه‌نویسی

در صورتیکه آخرین گره در یک لیست (یا دیگر ساختمان‌های داده خطی) با **null** *تنظیم نشود با خطای منطقی مواجه خواهید شد.*

21-3 اخذ دینامیکی حافظه و ساختمان داده

ایجاد و نگهداری ساختمان‌های داده دینامیکی مستلزم اخذ دینامیکی حافظه است، قابلیت بدست آوردن حافظه بیشتر (برای نگهداری متغیرهای جدید) و آزاد کردن حافظه‌ای که در زمان اجرای برنامه دیگر به آن نیازی نیست. اخذ دینامیکی حافظه محدود به حافظه فیزیکی موجود در کامپیوتر است (و مقدار فضای موجود بر روی دیسک در حافظه مجازی سیستم). در بیشتر موارد، حافظه موجود در کامپیوتر بایستی مابین برنامه‌های دیگر به اشتراک گذاشته شود.

عملگر**new** برای اخذ حافظه دینامیکی لازم است. کلمه کلیدی **new** نام کلاسی را بعنوان یک آرگومان دریافت می‌کند. سپس بصورت دینامیکی، حافظه را برای شی جدید اخذ کرده، و اشاره‌گری به شی جدیداً ایجاد شده برگشت می‌دهد. برای مثال، عبارت:

```
Node *newPtr = new Node (10); //create Node with data 10
```

مقدار حافظه متناسب را برای ذخیره سازی **Node** به اجرا گذاشته، سازنده **Node** اجرا شده و آدرس شی جدید **Node** به **newPtr** تخصیص داده می‌شود. اگر حافظه در دسترس نباشد، **new** یک استثناء از نوع **bad_alloc** به راه می‌اندازد. مقدار 10 به سازنده **Node** ارسال شده و عضو داده **Node** با 10مقداردهی اولیه می‌شود.

عملگر **delete** مبادرت به اجرای نابود کننده **Node** کرده و حافظه اخذ شده با **new** را بازپس می‌گیرد. حافظه به سیستم برگردانده می‌شود. برای آزاد کردن حافظه‌ای که به روش دینامیکی در عبارت فوق اخذ شده است، از عبارت زیر استفاده می‌شود

```
delete newPtr;
```

دقت کنید که خود **newPtr** حذف نمی‌شود، بجای آن فضایی که **newPtr** به آن اشاره می‌کند حذف می‌گردد. اگر اشاره‌گر **newPtr** دارای اشاره‌گر **null** با مقدار صفر باشد، عبارت فوق تاثیری نخواهد داشت. حذف اشاره‌گر **null** خطا نیست.

در بخش‌های بعدی به توضیح لیست‌ها، پشته‌ها، صف‌ها و درخت‌ها می‌پردازیم. این ساختمان‌های داده با اخذ حافظه دینامیکی و کلاس‌های خود ارجاعی ایجاد و نگهداری می‌شوند.

21-4 لیست‌های پیوندی

یک *لیست پیوندی* (*Linked list*) مجموعه خطی، از شی‌های کلاس خود ارجاعی به نام گره است که توسط لینک‌های اشاره‌گر به یکدیگر متصل شده‌اند. یک لیست پیوندی از طریق اشاره‌گر به اولین گره لیست قابل دسترس است. گره‌های بعدی از طریق قسمت لینک اشاره‌گر که در هر گره ذخیره شده‌اند، قابل دسترس‌اند. طبق قاعده لینک اشاره‌گر آخرین گره لیست با **null** تنظیم می‌شود که انتهای لیست را نشان می‌دهد. داده‌ها در لیست پیوندی بصورت دینامیکی ذخیره می‌شوند و هرگره زمانی ایجاد می‌شود که به آن نیاز است. یک گره می‌تواند شامل هر نوع داده‌ای از جمله شی‌هایی از سایر کلاس‌ها باشد. پشته‌ها و صف‌ها جزء ساختمان داده‌های خطی هستند و همانطوری که خواهید دید، آنها نوعی لیست پیوندی می‌باشند، در حالیکه درخت‌ها جزء داده‌های غیرخطی هستند.

اگر چه داده‌ها را می‌توان در آرایه‌ها ذخیره کرد، اما لیست‌های پیوندی مزیت‌های نسبت به آرایه‌ها دارند، از جمله هنگامی که تعداد عناصر داده مشخص نباشد، کاربرد لیست‌ها بسیار مناسب است. لیست‌های پیوندی حالت دینامیکی دارند و از اینرو طول لیست می‌تواند در صورت نیاز افزایش یا کاهش

یابد. در حالیکه در آرایه‌ها با سایز ثابت چنین کاری نمی‌توان انجام داد چرا که سایز آرایه در هنگام ایجاد تعیین می‌شود. آرایه می‌تواند پر شود اما لیست‌های پیوندی فقط در زمانیکه سیستم با کمبود حافظه مواجه شود و نتواند حافظه‌ای در اختیار لیست قرار دهد، پر می‌شوند. برنامه‌نویسان می‌توانند با وارد کردن هر عنصر جدید در مکان مناسب در لیست، همواره لیست پیوندی را بصورت مرتب شده نگهداری کنند. اگر چه انجام اینکار زمانبر است اما نیازی به جابجا کردن عناصر لیست ندارد.

کارآیی

عمل درج و حذف در یک آرایه مرتب شده زمانگیر است چرا که تمام عضوهای موجود در آرایه بایستی بصورت صحیح جابجا شوند.

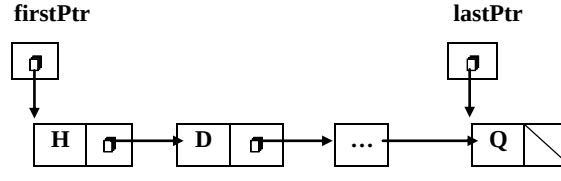
کارآیی

عناصر آرایه بصورت به هم پیوسته در حافظه ذخیره می‌شوند. از اینرو می‌توان بلافاصله به عنصر مورد نظر در آرایه با توجه به آدرس شروع آرایه دسترسی پیدا کرد، در حالیکه در لیست‌های پیوندی نمی‌توان چنین دسترسی داشت.

معمولاً گره‌های لیست پیوندی در حافظه پشت سرهم و متصل به یکدیگر ذخیره نمی‌شود. در عوض گره‌ها بصورت منطقی در پشت سرهم قرار دارند. شکل 2-21 نشاندهنده یک لیست پیوندی متشکل از چند گره است.

پیاده‌سازی لیست پیوندی

در برنامه‌های 3-21 الی5-21 از یک الگوی کلاس **List** برای کار با لیستی از مقادیر صحیح و لیستی از مقادیر اعشاری است. برنامه راه‌انداز (شکل 5-21) پنج گزینه دارد: 1) مقداری به ابتدای لیست اضافه می‌کند، 2) مقداری به انتهای لیست اضافه می‌کند،3) مقداری را از ابتدای لیست حذف می‌کند،4) مقداری را از انتهای لیست حذف می‌کند و 5) پردازش لیست را پایان می‌دهد.



شکل 2-21 | نمایش گرافیکی از یک لیست پیوندی.

برنامه از الگوهای کلاس **ListNode** (شکل3-21) و **List** (شکل4-21) استفاده می‌کند.کپسوله کردن در هر شی **List**، یک لیست پیوندی از شی‌های **ListNode** است. کلاس **ListNode** (شکل3-21)

حاوی دو عضو داده **data** و **nextPtr** (خطوط 20-19)، یک سازنده برای مقداردهی این اعضا و تابع **getData** برای برگشت دادن داده در گره است. عضو **data** مقداری از نوع **NODETYPE**، نوع پارامتر ارسالی به الگوی کلاس را ذخیره می‌کند. عضو **nextPtr** مبادرت به ذخیره کردن اشاره‌گر به شی بعدی **ListNode** در لیست پیوندی می‌کند.

خطوط 25-24 از کلاس **List**، شکل 4-21، متشکل از اعضای داده **firstPtr** (اشاره کننده به اولین **ListNode** در یک **List**) و **lastNode** (اشاره کنند. به آخرین **ListNode** در یک **List**) است. سازنده پیش‌فرض (خطوط 37-32) مبادرت به مقداردهی اولیه هر دو اشاره‌گر با **null** می‌کند. نابود کننده در خطوط 60-40 ما را مطمئن می‌کند که تمام شی‌های **ListNode** در یک شی **List** زمانیکه شی **List** حذف می‌شود، نابود می‌شوند. توابع اصلی **List** عبارتند از **insertAtfront** (خطوط75-63)، **insertAtBack** (خطوط 90-78)، **removeFromFront** (خطوط 111-93) و **removeFromBack** (خطوط 141-114).

```
1  // Fig. 21.3: Listnode.h
2  // Template ListNode class definition.
3  #ifndef LISTNODE_H
4  #define LISTNODE_H
5
6  // forward declaration of class List required to announce that class
7  //List exists so it can be used in the friend declaration at line 13
8  template< typename NODETYPE > class List;
9
10 template< typename NODETYPE >
11 class ListNode
12 {
13    friend class List< NODETYPE >; // make List a friend
14
15 public:
16    ListNode( const NODETYPE & ); // constructor
17    NODETYPE getData() const; // return data in node
18 private:
19    NODETYPE data; // data
20    ListNode< NODETYPE > *nextPtr; // next node in list
21 }; // end class ListNode
22
23 // constructor
24 template< typename NODETYPE >
25 ListNode< NODETYPE >::ListNode( const NODETYPE &info )
26    : data( info ), nextPtr( 0 )
27 {
28    // empty body
29 } // end ListNode constructor
30
31 // return copy of data in node
32 template< typename NODETYPE >
33 NODETYPE ListNode< NODETYPE >::getData() const
34 {
35    return data;
36 } // end function getData
```

```
37
38 #endif
```

شکل 3-21 | تعریف کلاس ListNode.

```
1  // Fig. 21.4: List.h
2  // Template List class definition.
3  #ifndef LIST_H
4  #define LIST_H
5
6  #include <iostream>
7  using std::cout;
8
9  #include "Listnode.h" // ListNode class definition
10
11 template< typename NODETYPE >
12 class List
13 {
14 public:
15    List(); // constructor
16    ~List(); // destructor
17    void insertAtFront( const NODETYPE & );
18    void insertAtBack( const NODETYPE & );
19    bool removeFromFront( NODETYPE & );
20    bool removeFromBack( NODETYPE & );
21    bool isEmpty() const;
22    void print() const;
23 private:
24    ListNode< NODETYPE > *firstPtr; // pointer to first node
25    ListNode< NODETYPE > *lastPtr; // pointer to last node
26
27    // utility function to allocate new node
28    ListNode< NODETYPE > *getNewNode( const NODETYPE & );
29 }; // end class List
30
31 // default constructor
32 template< typename NODETYPE >
33 List< NODETYPE >::List()
34    : firstPtr( 0 ), lastPtr( 0 )
35 {
36    // empty body
37 } // end List constructor
38
39 // destructor
40 template< typename NODETYPE >
41 List< NODETYPE >::~List()
42 {
43    if ( !isEmpty() ) // List is not empty
44    {
45       cout << "Destroying nodes ...\n";
46
47       ListNode< NODETYPE > *currentPtr = firstPtr;
48       ListNode< NODETYPE > *tempPtr;
49
50       while ( currentPtr != 0 ) // delete remaining nodes
51       {
52          tempPtr = currentPtr;
53          cout << tempPtr->data << '\n';
54          currentPtr = currentPtr->nextPtr;
55          delete tempPtr;
56       } // end while
57    } // end if
```

```
58
59    cout << "All nodes destroyed\n\n";
60 } // end List destructor
61
62 // insert node at front of list
63 template< typename NODETYPE >
64 void List< NODETYPE >::insertAtFront( const NODETYPE &value )
65 {
66    ListNode< NODETYPE > *newPtr = getNewNode( value ); // new node
67
68    if ( isEmpty() ) // List is empty
69       firstPtr = lastPtr = newPtr; // new list has only one node
70    else // List is not empty
71    {
72     newPtr->nextPtr = firstPtr;//point new node to previous 1st node
73       firstPtr = newPtr; // aim firstPtr at new node
74    } // end else
75 } // end function insertAtFront
76
77 // insert node at back of list
78 template< typename NODETYPE >
79 void List< NODETYPE >::insertAtBack( const NODETYPE &value )
80 {
81    ListNode< NODETYPE > *newPtr = getNewNode( value ); // new node
82
83    if ( isEmpty() ) // List is empty
84       firstPtr = lastPtr = newPtr; // new list has only one node
85    else // List is not empty
86    {
87       lastPtr->nextPtr = newPtr; // update previous last node
88       lastPtr = newPtr; // new last node
89    } // end else
90 } // end function insertAtBack
91
92 // delete node from front of list
93 template< typename NODETYPE >
94 bool List< NODETYPE >::removeFromFront( NODETYPE &value )
95 {
96    if ( isEmpty() ) // List is empty
97       return false; // delete unsuccessful
98    else
99    {
100     ListNode<NODETYPE> *tempPtr = firstPtr;//hold tempPtr to delete
101
102      if ( firstPtr == lastPtr )
103         firstPtr = lastPtr = 0; // no nodes remain after removal
104      else
105         firstPtr = firstPtr->nextPtr; // point to previous 2nd node
106
107      value = tempPtr->data; // return data being removed
108      delete tempPtr; // reclaim previous front node
109      return true; // delete successful
110   } // end else
111 } // end function removeFromFront
112
113 // delete node from back of list
114 template< typename NODETYPE >
115 bool List< NODETYPE >::removeFromBack( NODETYPE &value )
116 {
117    if ( isEmpty() ) // List is empty
118       return false; // delete unsuccessful
119    else
```

```
119    {
120      ListNode<NODETYPE> *tempPtr = lastPtr;//hold tempPtr to delete
121
122       if ( firstPtr == lastPtr ) // List has one element
123          firstPtr = lastPtr = 0; // no nodes remain after removal
124       else
125       {
126          ListNode< NODETYPE > *currentPtr = firstPtr;
127
128          // locate second-to-last element
129          while ( currentPtr->nextPtr != lastPtr )
130             currentPtr = currentPtr->nextPtr; // move to next node
131
132          lastPtr = currentPtr; // remove last node
133          currentPtr->nextPtr = 0; // this is now the last node
134       } // end else
135
136       value = tempPtr->data; // return value from old last node
137       delete tempPtr; // reclaim former last node
138       return true; // delete successful
139    } // end else
140 } // end function removeFromBack
141
142 // is List empty?
143 template< typename NODETYPE >
144 bool List< NODETYPE >::isEmpty() const
145 {
146    return firstPtr == 0;
147 } // end function isEmpty
148
149 // return pointer to newly allocated node
150 template< typename NODETYPE >
151 ListNode< NODETYPE > *List< NODETYPE >::getNewNode(
152    const NODETYPE &value )
153 {
154    return new ListNode< NODETYPE >( value );
155 } // end function getNewNode
156
157 // display contents of List
158 template< typename NODETYPE >
159 void List< NODETYPE >::print() const
160 {
161    if ( isEmpty() ) // List is empty
162    {
163       cout << "The list is empty\n\n";
164       return;
165    } // end if
166
167    ListNode< NODETYPE > *currentPtr = firstPtr;
168
169    cout << "The list is: ";
170
171    while ( currentPtr != 0 ) // get element data
172    {
173       cout << currentPtr->data << ' ';
174       currentPtr = currentPtr->nextPtr;
175    } // end while
176
177    cout << "\n\n";
178 } // end function print
179
180 #endif
```

شکل 4–21 | تعریف کلاس List.

```
1  // Fig. 21.5: Fig21_05.cpp
2  // List class test program.
3  #include <iostream>
4  using std::cin;
5  using std::cout;
6  using std::endl;
7
8  #include <string>
9  using std::string;
10
11 #include "List.h" // List class definition
12
13 // function to test a List
14 template< typename T >
15 void testList( List< T > &listObject, const string &typeName )
16 {
17    cout << "Testing a List of " << typeName << " values\n";
18    instructions(); // display instructions
19
20    int choice; // store user choice
21    T value; // store input value
22
23    do // perform user-selected actions
24    {
25       cout << "? ";
26       cin >> choice;
27
28       switch ( choice )
29       {
30          case 1: // insert at beginning
31             cout << "Enter " << typeName << ": ";
32             cin >> value;
33             listObject.insertAtFront( value );
34             listObject.print();
35             break;
36          case 2: // insert at end
37             cout << "Enter " << typeName << ": ";
38             cin >> value;
39             listObject.insertAtBack( value );
40             listObject.print();
41             break;
42          case 3: // remove from beginning
43             if ( listObject.removeFromFront( value ) )
44                cout << value << " removed from list\n";
45
46             listObject.print();
47             break;
48          case 4: // remove from end
49             if ( listObject.removeFromBack( value ) )
50                cout << value << " removed from list\n";
51
52             listObject.print();
53             break;
54       } // end switch
55    } while ( choice != 5 ); // end do...while
56
57    cout << "End list test\n\n";
58 } // end function testList
59
60 // display program instructions to user
```

```
61 void instructions()
62 {
63    cout << "Enter one of the following:\n"
64       << "  1 to insert at beginning of list\n"
65       << "  2 to insert at end of list\n"
66       << "  3 to delete from beginning of list\n"
67       << "  4 to delete from end of list\n"
68       << "  5 to end list processing\n";
69 } // end function instructions
70
71 int main()
72 {
73    // test List of int values
74    List< int > integerList;
75    testList( integerList, "integer" );
76
77    // test List of double values
78    List< double > doubleList;
79    testList( doubleList, "double" );
80    return 0;
81 } // end main
```

```
Testing a list of integer values
Enter one of the following:
 1 to insert at beginning of list
 2 to insert at end of list
 3 to delete from beginning of list
 4 to delete from end of list
 5 to end list processing
? 1
Enter integer: 1
The list is: 1

? 1
Enter integer: 2
The list is: 2 1

? 2
Enter integer: 3
The list is: 2 1 3

? 2
Enter integer: 4
The list is: 2 1 3 4

? 3
2 remove from list
The list is: 1 3 4

? 3
1 remove from list
The list is:  3 4

? 4
4 remove from list
The list is: 4

? 4
3 remove from list
The list is empty

? 5
End list test
```

```
Testing a List of double values
Enter one of the following:
 1 to insert at beginning of list
 2 to insert at end of list
 3 to delete from beginning of list
 4 to delete from end of list
 5 to end list processing
? 1
Enter double: 1.1
The list is: 1.1

? 1
Enter double: 2.2
The list is: 2.2 1.1

? 2
Enter double: 3.3
The list is: 2.2 1.1 3.3

? 2
Enter double: 4.4
The list is: 2.2 1.1 3.3 4.4

? 3
2.2 removed from list
The list is: 1.1 3.3 4.4

? 3
1.1 removed from list
The list is: 3.3 4.4

? 4
4.4 removed from list
The list is: 3.3

? 4
3.3 removed from list
The list is empty

? 5
End list test

All nodes destroyed
All nodes destroyed
```

شکل 5-21 | کار با لیست پیوندی.

تابع **isEmpty** (خطوط 148-144) یک *تابع پیشگو* است که تعیین می‌کند آیا لیست تهی است یا خیر. اگر لیست تهی باشد، تابع **isEmpty** مقدار **true** و در غیر اینصورت **false** باز می‌گرداند. تابع **print** (خطوط 179-159) محتویات لیست را به نمایش در می‌آورد. تابع کمکی **getNewNode** در خطوط 151- 156 یک شی **ListNode** اخذ شده به روش دینامیکی را برگشت می‌دهد. این تابع از سوی توابع **insertAtFront** و **insertAtBack** فراخوانی می‌شود.

برنامه راه‌انداز (شکل 5-21) از تابع **testList** برای اینکه کاربر بتواند با شی‌های کلاس **List** کار کند استفاده کرده است. خطوط 74 و 78 شی‌های لیست را برای نوع‌های **int** و **double** ایجاد می‌کنند. خطوط 75 و 79 تابع **testList** را به همراه این شی‌های **List** فراخوانی می‌کنند.

تابع عضو insertAtFront

در چند صفحه بعدی در مورد عملکرد هر کدامیک از توابع عضو کلاس **List** توضیح خواهیم داد. تابع **insertAtFront** (شکل 4-21، خطوط 75-63) یک گره جدید در ابتدای لیست قرار می‌دهد. این تابع در چند مرحله عمل می‌کند که بصورت زیر می‌توان آنرا توضیح داد (به شکل 6-21 توجه کنید):

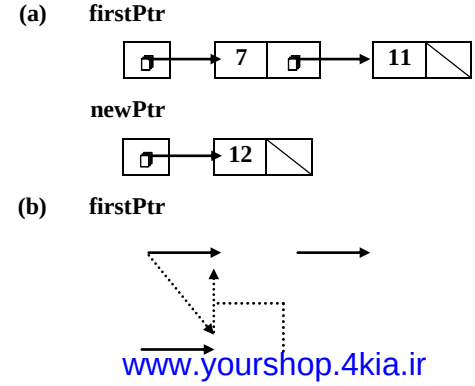
1- فراخوانی تابع **getNewNode**، ارسال مقدار به آن، که یک مراجعه ثابت به مقدار گره‌ای است که درج می‌شود (خط 66).

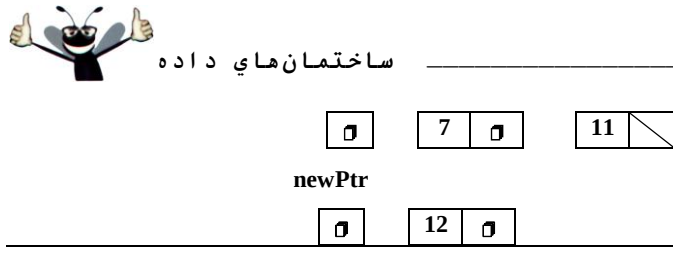
2- تابع **getNewNode** در خطوط 156-151 از عملگر **new** برای ایجاد یک گره جدید و برگشت دادن یک اشاره‌گر به گره جدیدا اخذ شده استفاده می‌کند، که به **newPtr** در **insertAtFront** تخصیص داده می‌شود (خط 66).

3- اگر لیست تهی باشد، هم **firstPtr** و هم **lastPtr** با **newPtr** تنظیم خواهند شد (خط 69).

4- اگر لیست خالی نباشد (خط 70)، گره مورد اشاره توسط **newPtr** به لیست با کپی **firstPtr** به **newPtr->nextPtr** متصل می‌شود (خط 72)، از اینرو است که گره جدید به اولین گره لیست اشاره می‌کند، و با کپی کردن **newPtr** به **firstPtr** (خط 73) است که **firstPtr** به اولین گره جدید در لیست اشاره می‌کند.

شکل 6-21 نحوه عملکرد تابع **insertAtFront** را نشان می‌دهد. بخش (a) در حال نمایش لیست و گره جدیدی است که در عملیات **insertAtFront** بوجود آمده و قبل از وصل کردن گره جدید (حاوی مقدار **12**) به لیست است. فلش‌های نقطه‌چین در بخش (b) مراحل عملکرد **insertAtFront** را که در مرحله 4 شرح داده شده است و در تبدیل به گره اول در لیست می‌شود را نشان‌می‌دهند.





شکل 6-21 | نمایش گرافیکی از عملکرد insertAtFront.

تابع عضو insertAtBack

تابع **insertAtBack** (شکل 4-21، خطوط 90-78) یک گره جدید در انتهای لیست قرار می‌دهد. عملکرد این تابع در چهار مرحله است (شکل 7-21):

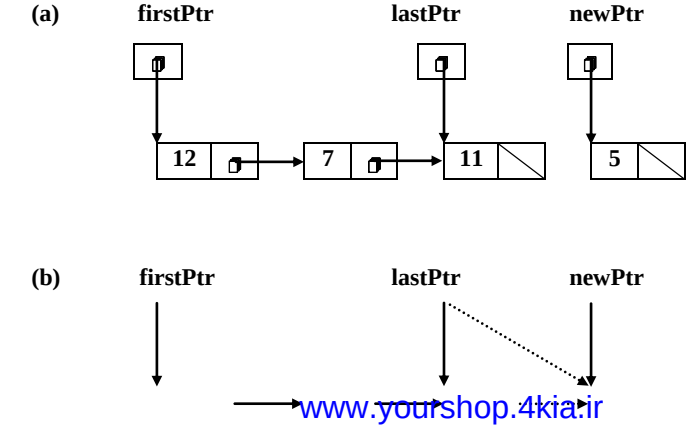
1- فراخوانی تابع **getNewNode**، ارسال مقدار به آن، که یک مراجعه ثابت به مقدار گره‌ای است که درج می‌شود (خط 81).

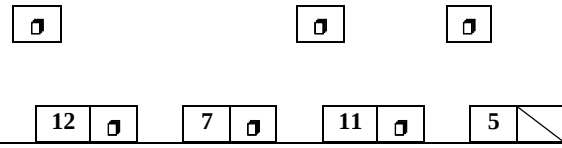
2- تابع **getNewNode** در خطوط 156-151 از عملگر new برای ایجاد یک گره جدید و برگشت دادن یک اشاره‌گر به گره جدیدا اخذ شده استفاده می‌کند، که به **newPtr** در **insertAtBack** تخصیص داده می‌شود (خط 81).

3- اگر لیست تهی باشد، هم **firstPtr** و هم **lastPtr** با **newPtr** تنظیم خواهند شد (خط 83).

4- اگر لیست خالی نباشد (خط 85)، گره مورد اشاره توسط **newPtr** به لیست با کپی **newPtr** به **lastPtr->nextPtr** متصل می‌شود (خط 87)، از اینرو است که گره جدید به آخرین گره لیست اشاره می‌کند، و با کپی کردن **newPtr** به **lastPtr** (خط 88) است که **lastPtr** به آخرین گره جدید در لیست اشاره می‌کند.

شکل 7-21 مراحل عملکرد تابع **insertAtBack** را نشان می‌دهد. بخش (a) در حال نمایش لیست و گره جدید (با مقدار 5) بوجود آمده در عملیات **insertBack** و قبل از متصل شدن گره به لیست است. فلش‌های نقطه‌چین در بخش (b) مراحلی که تابع **insertAtBack** طی می‌کند تا گره جدید به انتهای لیست افزوده شود، را نشان می‌دهند.





شکل 7-21 | نمایش گرافیکی ازعملکرد insertAtBack.

تابع عضو removeFromFront

تابع **removeFromFont** (شکل4-21،خطوط 111-93) مبادرت به حذف گره ابتدائی از لیست کرده و اشاره‌گری به داده حذف شده باز می‌گرداند. اگر برنامه سعی در حذف گره از یک لیست تهی نماید، تابع مقدار false و در غیر اینصورت مقدار true برگشت می‌دهد (خطوط 97-96). این تابع در شش مرحله وظیفه خود را به انجام می‌رساند:

1- تخصیص آدرس **tempPtr** به جای که firstPtr اشاره می‌کند (خط 100). سرانجام، tempPtr برای حذف گره بکار گرفته می‌شود.

2- اگر **firstPtr** و **lastPtr** یکی باشند (خط 102) پس لیست حاوی یک عنصر است. در این حالت، تابع مبادرت به تنظیم **firstPtr** و **lastPtr** با **0** (خط103) می‌کند، تا گره از لیست حذف شود (لیست تهی می‌شود).

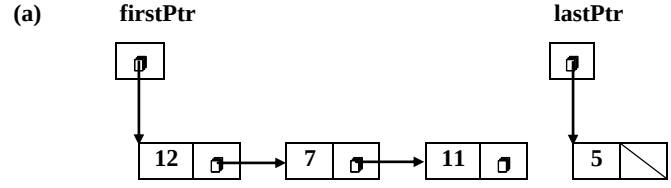
3- اگر لیست حاوی بیش از یک گره باشد، گره اولی حذف خواهد شد، در اینحالت اشاره‌گر **lastPtr** در جای خود باقی می‌ماند و فقط **firstPtr** با **firstPtr->nextPtr** تنظیم می‌شود(خط105). از اینرو، **firstPtr** به گره دوم قبل از فراخوانی تابع **removeFromFront** اشاره می‌کند.

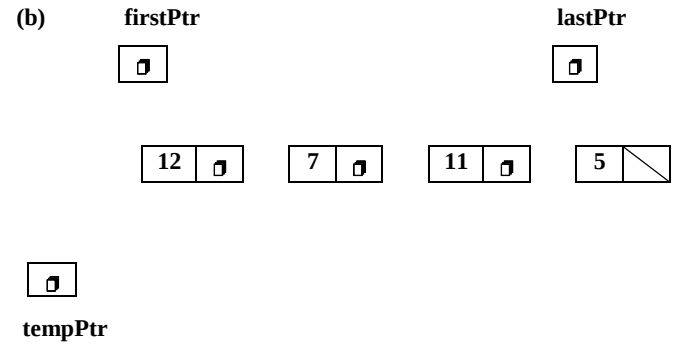
4- پس از اینکه کار تمام این اشاره‌گرها تمام شد، مقدار عضو **data** متعلق به گره‌ای که حذف خواهد شد، به **value** تخصیص داده می‌شود (خط 107).

5- حال گره مورد اشاره توسط **tempPtr** حذف می‌شود (خط 108).

6- برگشت **true**، نشان می‌دهد که حذف با موفقیت صورت گرفته است (خط 109).

شکل 8-21مراحل عملکرد تابع **removeFromFront** را نشان می‌دهد. بخش (a) در حال نمایش لیست قبل از انجام عمل حذف است. بخش (b) نحوه جابجایی اشاره‌گر را نشان می‌دهد.





شکل 8-21 | نمایش گرافیکی عملکرد removeFromFront.

تابع عضو removeFromBack

تابع **removeFromBack** (شکل4-21،خطوط141-114) آخرین گره از لیست را حذف و مراجعه‌ای به داده حذف شده باز می‌گرداند. اگر برنامه مبادرت به حذف گره از یک لیست تهی نماید، تابع مقدار **false** (خطوط 118-117) و در غیر اینصورت مقدار **true** برگشت خواهد داد. عملکرد این تابع در نه مرحله صورت می‌گیرد(شکل 9-21):

1- تخصیص آدرس **tempPtr** به جای که **lastPtr** اشاره می‌کند (خط 121). سرانجام، **tempPtr** برای حذف گره بکار گرفته می‌شود..

2- اگر **firstPtr** و **lastPtr** یکی باش‌ند (خط 123) پس لیست حاوی یک عنصر است. در این حالت، تابع مبادرت به تنظیم **firstPtr** و **lastPtr** با **0** (خط124) می‌کند، تا گره از لیست حذف شود (لیست تهی می‌شود).

3- اگر لیست حاوی بیش از یک گره باشد، اشاره‌گر **currentPtr** ایجاد و به **firstPtr** تخصیص داده می‌شود (خط 127).

4- از **currentPtr** برای پیمایش لیست تا رسیدن آن به گره ماقبل آخر استفاده می‌شود. حلقه **while** (خطوط 131-130) تا مادامیکه **currentPtr->nextPtr** برابر **lastPtr** نشده، آنرا به **currentPtr** اشاره می‌دهد.

5- پس از یافتن گره ما قبل آخر، **currentPtr** به **lastPtr** تخصیص یافته (خط 133) و موجب می‌شود تا گره آخر از لیست جدا شود.

6- تنظیم **currentPtr->nextPtr** گره جدید آخر لیست با **0** (خط 134).

7- پس از اینکه کار تمام این اشاره‌گرها تمام شد، مقدار عضو **data** متعلق به گره‌ای که حذف خواهد شد، به **value** تخصیص داده می‌شود (خط 137).

8- حال گره مورد اشاره توسط **tempPtr** حذف می‌شود (خط 138).

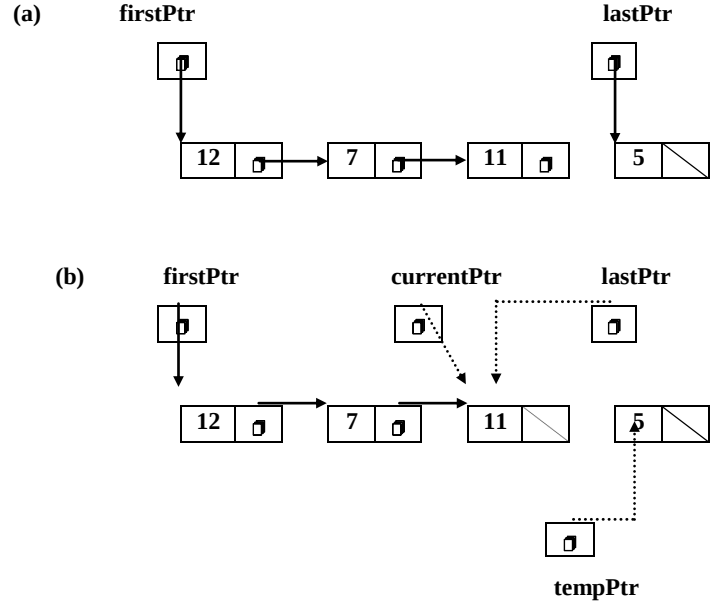
9- برگشت **true**، نشان می‌دهد که حذف با موفقیت صورت گرفته است (خط 139).

شکل 9-21 مراحل عملکرد تابع **removeFromBack** را نشان می‌دهد. بخش (a) در حال نمایش لیست قبل از انجام عمل حذف است. بخش (b) نحوه جابجایی اشاره‌گر را نشان می‌دهد.

23-5 پشته‌ها

پشته‌ها فضاهای برای متغیرهای محلی به هنگام فراخوانی روال ایجاد می‌کنند. هنگامی که روال به، روال فراخوان خود باز می‌گردد، فضای اخذ شده برای متغیرها از پشته حذف می‌شود و دیگر متغیرها در برنامه شناخته نمی‌شوند.همچنین از پشته‌ها برای ارزیابی عبارات محاسباتی در کامپایلرها و ایجاد کد زبان ماشین برای این عبارات استفاده می‌شود.

با استفاده از مزایای لیست‌ها و پشته‌ها، کلاس پشته مورد نظر خود را پیاده‌سازی می‌کنیم (با استفاده مجدد از کلاس لیست). به توضیح دو نوع متفاوت از بکارگیری استفاده مجدد می‌پردازیم. ابتدا کلاس پشته را با ارث‌بری از کلاس **List** پیاده‌سازی می‌کنیم. سپس همان کلاس پشته را از طریق ترکیب با بکارگیری یک شی **List** بعنوان یک عضو **private** در کلاس پشته پیاده‌سازی خواهیم کرد.



شکل 9-21 | نمایش گرافیکی عملکرد removeFromBack.

برنامه‌های شکل 13-21 و 14-21 یک کلاس پشته (شکل 13-21) را از طریق ارث‌بری (خط 9) از کلاس **List** موجود در برنامه 4-21 ایجاد می‌کنند. می‌خواهیم که پشته دارای توابع **push** (خطوط 13-16)، **pop** (خطوط 19-22)، **isStackEmpty** (خطوط 25-28) و **printStack** (خطوط 31-34) باشد. ضرورتاً، آنها توابع **insertAtFront**، **removeFromFront**، **isEmpty** و **print** از کلاس **List** خواهند بود. کلاس **List** حاوی کلاس‌های دیگری همانند **insertAtBack** و **removeFromBack** است که مایل نیستیم از طریق واسط **public** پشته در دسترس قرار گیرند.

به هنگام پیاده‌سازی تابعهای پشته، تابعهای متناسب در **List** فراخوانی خواهند شد، تابع **push** تابع **insertAtFront** (خط 15)، تابع **pop** تابع **removeFromFront** (خط 21) و تابع **isStackEmpty** تابع **isEmpty** (خط 27) و تابع **printStack** تابع **print** (خط 33) را فراخوانی می‌کند.

```
1  // Fig. 21.13: Stack.h
2  // Template Stack class definition derived from class List.
3  #ifndef STACK_H
4  #define STACK_H
5
6  #include "List.h" // List class definition
7
8  template< typename STACKTYPE >
9  class Stack : private List< STACKTYPE >
10 {
11 public:
12    // push calls the List function insertAtFront
13    void push( const STACKTYPE &data )
14    {
15       insertAtFront( data );
16    } // end function push
17
18    // pop calls the List function removeFromFront
19    bool pop( STACKTYPE &data )
20    {
21       return removeFromFront( data );
22    } // end function pop
23
24    // isStackEmpty calls the List function isEmpty
25    bool isStackEmpty() const
26    {
27       return isEmpty();
28    } // end function isStackEmpty
29
30    // printStack calls the List function print
31    void printStack() const
32    {
33       print();
34    } // end function print
35 }; // end class Stack
36
37 #endif
```

شکل 13-21 | تعریف کلاس پشته.

کلاس پشته در **main** (شکل 14-21) برای نمونه‌سازی پشته صحیح **intStack** از نوع **Stack<int>** بکار گرفته شده است (خط 11). مقادیر صحیح 0 تا 2 وارد پشته **intStack** شده (خطوط 20-16)، سپس از آن خارج می‌شوند (خطوط 30-25). برنامه از کلاس پشته برای ایجاد یک **doubleStack** از نوع **Stack<double>** استفاده کرده است (خط 32). مقادیر 2.2 ,1.1 و 3.3 وارد **doubleStack** شده (خطوط 43-38) و سپس از آن خارج شده‌اند (خطوط 53-48).

```
1  // Fig. 21.14: Fig21_14.cpp
2  // Template Stack class test program.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Stack.h" // Stack class definition
8
9  int main()
10 {
11    Stack< int > intStack; // create Stack of ints
12
13    cout << "processing an integer Stack" << endl;
14
15    // push integers onto intStack
16    for ( int i = 0; i < 3; i++ )
17    {
18       intStack.push( i );
19       intStack.printStack();
20    } // end for
21
22    int popInteger; // store int popped from stack
23
24    // pop integers from intStack
25    while ( !intStack.isStackEmpty() )
26    {
27       intStack.pop( popInteger );
28       cout << popInteger << " popped from stack" << endl;
29       intStack.printStack();
30    } // end while
31
32    Stack< double > doubleStack; // create Stack of doubles
33    double value = 1.1;
34
35    cout << "processing a double Stack" << endl;
36
37    // push floating-point values onto doubleStack
38    for ( int j = 0; j < 3; j++ )
39    {
40       doubleStack.push( value );
41       doubleStack.printStack();
42       value += 1.1;
43    } // end for
44
45    double popDouble; // store double popped from stack
46
47    // pop floating-point values from doubleStack
48    while ( !doubleStack.isStackEmpty() )
49    {
```

```
50       doubleStack.pop( popDouble );
51       cout << popDouble << " popped from stack" << endl;
52       doubleStack.printStack();
53    } // end while
54
55    return 0;
56 } // end main
```

```
processing an integer stack
The list is: 0

The list is: 1 0

The list is: 2 1 0

2 popped from stack
The list is: 1 0

1 popped from stack
The list is: 0

0 popped from stack
The list is empty

processing a double Stack
The list is: 1.1

The list is: 2.2 1.1

The list is: 3.3 2.2 1.1

3.3 popped from stack
The list is: 2.2 1.1

2.2 popped from stack
The list is: 1.1

1.1 popped from stack
The list is empty

All nodes destroyed

All nodes destroyed
```

شکل 14-21 | برنامه ساده پشته.

روش دیگر در پیاده‌سازی یک کلاس پشته از طریق استفاده مجدد از کلاس لیست و به کمک ترکیب است. کلاس موجود در برنامه شکل 15-21 پیاده‌سازی جدیدی از کلاس پشته است که حاوی یک شی **List<STACKTYPE>** بنام **stackList** می‌باشد (خط 38). این نسخه از کلاس پشته از کلاس List شکل 4-21 استفاده می‌کند. برای تست این کلاس، از برنامه راه‌انداز در شکل 14-21 استفاده شده است، اما حاوی فایل سرآیند Stackcomposition.h در خط 6 است. خروجی برنامه با هر دو نسخه کلاس پشته یکسان است.

```
1  // Fig. 21.15: Stackcomposition.h
2  // Template Stack class definition with composed List object.
3  #ifndef STACKCOMPOSITION_H
4  #define STACKCOMPOSITION_H
```

```
5
6  #include "List.h" // List class definition
7
8  template< typename STACKTYPE >
9  class Stack
10 {
11 public:
12    // no constructor; List constructor does initialization
13
14    // push calls stackList object's insertAtFront member function
15    void push( const STACKTYPE &data )
16    {
17       stackList.insertAtFront( data );
18    } // end function push
19
20    // pop calls stackList object's removeFromFront member function
21    bool pop( STACKTYPE &data )
22    {
23       return stackList.removeFromFront( data );
24    } // end function pop
25
26    // isStackEmpty calls stackList object's isEmpty member function
27    bool isStackEmpty() const
28    {
29       return stackList.isEmpty();
30    } // end function isStackEmpty
31
32    // printStack calls stackList object's print member function
33    void printStack() const
34    {
35       stackList.print();
36    } // end function printStack
37 private:
38    List< STACKTYPE > stackList; // composed List object
39 }; // end class Stack
40
41 #endif
```

شکل 15-21 | کلاس پشته با ترکیب شی لیست.

21-6 صف‌ها

نوع دیگری از ساختمان داده‌ها، *صف* (*Queue*) است. صف شبیه به یک صف انتظار (نوبت) در یک فروشگاه است، که در آن ابتدا به اولین شخص در سر صف، سرویس داده می‌شود و مشتری‌های بعدی باید در انتظار رسیدن نوبت خود باقی بمانند. در صف گره‌ها از ابتدا یا سر صف (head) حذف می‌شوند و اضافه کردن گره به صف از انتهای (tail) آن صورت می‌گیرد، به همین دلیل به ساختمان داده صف، اولین ورودی–اولین خروجی (*FIFO*) گفته می‌شود. به عمل افزودن گره به صف *enqueue* و حذف گره از صف *dequeue* گفته می‌شود.

در سیستم‌های کامپیوتری صف‌ها کاربردهای متفاوتی دارند. بیشتر کامپیوترها دارای یک پردازنده هستند، بنابراین فقط یک کاربرد در هر زمان می‌تواند سرویس بگیرد و تقاضای سایر کاربران در صف قرار می‌گیرد. همچنین از صف‌ها در فرآیند چاپ هم استفاده می‌شود. در یک محیط چند کاربره ممکن است، فقط یک چاپگر وجود داشته باشد و کاربران هم برای ارسال مستندات خود به چاپگر نیاز داشته

باشند. اگر چاپگر مشغول باشد، ممکن است خروجی‌های دیگری در حال تولید باشند و از اینرو یک صف انتظار برای دسترسی به چاپگر تشکیل می‌شود.

در شبکه‌های کامپیوتری بسته‌های اطلاعاتی در صف‌های انتظار قرار می‌گیرند و در هر زمان یک بسته به عنوان گره به شبکه ارسال می‌شود و اگر نیاز باشد تا رسیدن به مقصد این بسته جابجا می‌شود.

برنامه‌های شکل 16-21 و 17-21 کلاس صف (Queue) را از طریق ارث‌بری (خط 9) از کلاس List ایجاد می‌کنند (شکل 4-21). می‌خواهیم کلاس **Queue** حاوی توابع **enqueue** (خطوط 16-13)، **dequeue** (خطوط 22-19)، **isQueueEmpty** (خطوط 28-25) و **printQueue** (خطوط 34-31) باشد. دقت کنید که این توابع ضرورتاً تابعهای **insertAtBack**، **removeFromFront**، **isEmpty** و **print** از کلاس **List** هستند.

```
1  // Fig. 21.16: Queue.h
2  // Template Queue class definition derived from class List.
3  #ifndef QUEUE_H
4  #define QUEUE_H
5
6  #include "List.h" // List class definition
7
8  template< typename QUEUETYPE >
9  class Queue: private List< QUEUETYPE >
10 {
11 public:
12    // enqueue calls List member function insertAtBack
13    void enqueue( const QUEUETYPE &data )
14    {
15       insertAtBack( data );
16    } // end function enqueue
17
18    // dequeue calls List member function removeFromFront
19    bool dequeue( QUEUETYPE &data )
20    {
21       return removeFromFront( data );
22    } // end function dequeue
23
24    // isQueueEmpty calls List member function isEmpty
25    bool isQueueEmpty() const
26    {
27       return isEmpty();
28    } // end function isQueueEmpty
29
30    // printQueue calls List member function print
31    void printQueue() const
32    {
33       print();
34    } // end function printQueue
35 }; // end class Queue
36
37 #endif
```

شکل 16-21 | تعریف کلاس صف.

به هنگام پیاده‌سازی توابع صف، توابع متناسب در **List** فراخوانی خواهند شد، تابع **enqueue** تابع **insertAtBack** (خط 15)، تابع **dequeue** تابع **removeFromFront** (خط 21) و تابع **isQueueEmpty** تابع **isEmpty** (خط 27) و تابع **printQueue** تابع **print** (خط 33) را فراخوانی می‌کند.

کلاس صف در شکل 17-21 برای نمونه‌سازی صف صحیح **intQueue** از نوع **Queue<int>** بکار گرفته شده است (خط 11). مقادیر صحیح 0 تا 2 وارد صف **intQueue** شده (خطوط 16-20)، سپس از آن خارج می‌شوند (خطوط 25-30). برنامه از کلاس صف برای ایجاد یک **doubleQueue** از نوع **Queue<double>** استفاده کرده است (خط 32). مقادیر 2.2 ,1.1 و 3.3 وارد **doubleQueue** شده (خطوط 38-43) و سپس از آن خارج شده‌اند (خطوط 48-53).

```
1  // Fig. 21.17: Fig21_17.cpp
2  // Template Queue class test program.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Queue.h" // Queue class definition
8
9  int main()
10 {
11    Queue< int > intQueue; // create Queue of integers
12
13    cout << "processing an integer Queue" << endl;
14
15    // enqueue integers onto intQueue
16    for ( int i = 0; i < 3; i++ )
17    {
18       intQueue.enqueue( i );
19       intQueue.printQueue();
20    } // end for
21
22    int dequeueInteger; // store dequeued integer
23
24    // dequeue integers from intQueue
25    while ( !intQueue.isQueueEmpty() )
26    {
27       intQueue.dequeue( dequeueInteger );
28       cout << dequeueInteger << " dequeued" << endl;
29       intQueue.printQueue();
30    } // end while
31
32    Queue< double > doubleQueue; // create Queue of doubles
33    double value = 1.1;
34
35    cout << "processing a double Queue" << endl;
36
37    // enqueue floating-point values onto doubleQueue
38    for ( int j = 0; j < 3; j++ )
39    {
40       doubleQueue.enqueue( value );
41       doubleQueue.printQueue();
42       value += 1.1;
43    } // end for
```

```
44
45    double dequeueDouble; // store dequeued double
46
47    // dequeue floating-point values from doubleQueue
48    while ( !doubleQueue.isQueueEmpty() )
49    {
50       doubleQueue.dequeue( dequeueDouble );
51       cout << dequeueDouble << " dequeued" << endl;
52       doubleQueue.printQueue();
53    } // end while
54
55    return 0;
56 } // end main
```

```
processing an integer Queue
The list is:0

The list is: 0 1

The list is: 0 1 2

0 dequeued
The list is: 1 2

1 dequeued
The list is: 2

2 dequeued
The list is empty

processing an double Queue
The list is: 1.1

The list is: 1.1 2.2

The list is: 1.1 2.2 3.3

1.1 dequeued
The list is: 2.2 3.3

2.2 dequeued
The list is: 3.3

3.3 dequeued
The list is empty

All nodes destroyed

All nodes destroyed
```

شکل 17-21 | برنامه پردازش صف.

21-7 درخت‌ها

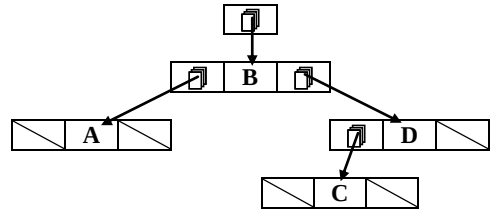
لیست‌های پیوندی، پشته‌ها و صف‌ها جزء *ساختمان داده‌های خطی* هستند (متوالی). در حالیکه درخت (tree) یک ساختمان داده خطی نیست و یک ساختمان داده دو بعدی با خصوصیات ویژه خود است. گره‌های درخت دارای دو یا بیشتر از دو لینک هستند. در این بخش در مورد درخت‌های باینری (دودوئی)

صحبت خواهیم کرد (شکل 18-21). در این نوع از درخت‌ها همه گره‌ها دارای دو لینک هستند (یکی یا هر دو می‌تواند برابر **null** باشد یا هیچ کدام).

گره ریشه اولین گره در درخت است. هر لینک گره ریشه، به یک فرزند اشاره می‌کند. فرزند چپ، اولین گره در زیر درخت چپ و فرزند راست اولین گره در زیر درخت راست است. به فرزندان یک گره sibling و به گره بدون فرزند گره برگ (leaf node) گفته می‌شود. دانشمندان کامپیوتر معمولاً درخت‌ها را از ریشه به پایین ترسیم می‌کنند که برخلاف رشد طبیعی درخت در طبیعت است.

درخت جستجوی باینری

در این بخش یک درخت باینری ویژه بنام *درخت جستجوی باینری* (*binary search tree*) ایجاد خواهیم کرد. یک درخت جستجوی باینری (با مقادیر غیر تکرار شونده در گره‌ها) دارای خصوصیاتی به شرح زیر است: مقادیر موجود در هر زیر درخت چپ کمتر از مقدار گره والد خود است و مقادیر موجود در هر زیر درخت راست بزرگتر از مقدار والد خود می‌باشد. شکل 19-21 یک درخت جستجوی باینری با 12 مقدار صحیح است.

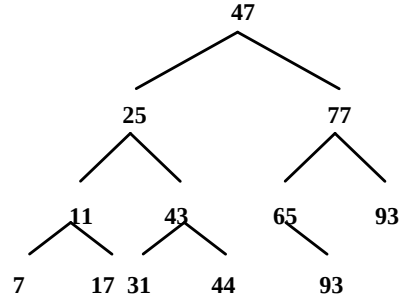


شکل 18-21 | نمایش گرافیکی از یک درخت باینری.

پیاده‌سازی درخت جستجوی باینری

برنامه شکل‌های 20-21 الی 22-21 یک درخت جستجوی باینری ایجاد و آن را به سه روش بازگشتی *inorder*، *preorder* و *postorder* پیمایش می‌کنند. برنامه مبادرت به ایجاد 10 عدد تصادفی کرده و هر یک را وارد درخت می‌کند. در شکل 17-23، کلاس **Tree** در فضای نامی **BinaryTreeLibrary** تعریف شده است. شکل 18-23 تعریف کننده کلاس **TreeTest** است که توصیف کننده قابلیت‌های **Tree** می‌باشد. تابع **Main** از مدول **TreeTest** یک شی تهی **Tree** نمونه‌سازی کرده، 10 مقدار صحیح تصادفی ایجاد و هر مقدار را در درخت باینری و با استفاده از تابع **InsertNode** وارد می‌کند. سپس برنامه پیمایش‌های preorder، inorder و postorder را بر روی درخت به انجام می‌رساند. در مورد هر کدامیک از این پیمایش‌ها توضیح خواهیم داد.

بحث خود را با برنامه راه‌انداز شکل 22-21 شروع کرده، سپس با پیاده‌سازی کلاس‌های **TreeNode** (شکل 20-21) و **Tree** (شکل 21-21) ادامه می‌دهیم. تابع **main** (شکل 22-21) با نمونه‌سازی درخت **intTree** از نوع **Tree<int>** شروع می‌شود (خط 15). برنامه ده مقدار صحیح درخواست کرده، هر یک را با فراخوانی تابع **insertNode** وارد درخت باینری می‌کند (خط 24). سپس برنامه پیمایش‌های preorder, inorder و postorder را بر روی **intTree** انجام می‌دهد (خطوط 31 ,28 و 34). در ادامه، برنامه مبادرت به نمونه‌سازی درخت **doubleTree** از نوع **Tree<double>** می‌کند (خط 36). برنامه ده مقدار **double** درخواست کرده، هر یک را با فراخوانی تابع **insertNode** وارد درخت باینری می‌کند (خط 46). سپس برنامه پیمایش‌های preorder, inorder و postorder را بر روی **doubleTree** انجام می‌دهد (خطوط 56 ,53 و 50).



شکل 19-21 | درخت جستجوی باینری.

```
1  // Fig. 21.20: Treenode.h
2  // Template TreeNode class definition.
3  #ifndef TREENODE_H
4  #define TREENODE_H
5
6  // forward declaration of class Tree
7  template< typename NODETYPE > class Tree;
8
9  // TreeNode class-template definition
10 template< typename NODETYPE >
11 class TreeNode
12 {
13    friend class Tree< NODETYPE >;
14 public:
15    // constructor
16    TreeNode( const NODETYPE &d )
17       : leftPtr( 0 ), // pointer to left subtree
18         data( d ), // tree node data
19         rightPtr( 0 ) // pointer to right substree
20    {
21       // empty body
22    } // end TreeNode constructor
23
24    // return copy of node's data
25    NODETYPE getData() const
```

```
26    {
27       return data;
28    } // end getData function
29 private:
30    TreeNode< NODETYPE > *leftPtr; // pointer to left subtree
31    NODETYPE data;
32    TreeNode< NODETYPE > *rightPtr; // pointer to right subtree
33 }; // end class TreeNode
34
35 #endif
```

شکل 20–21 | تعریف کلاس TreeNode.

```
1  // Fig. 21.21: Tree.h
2  // Template Tree class definition.
3  #ifndef TREE_H
4  #define TREE_H
5
6  #include <iostream>
7  using std::cout;
8  using std::endl;
9
10 #include <new>
11 #include "Treenode.h"
12
13 // Tree class-template definition
14 template< typename NODETYPE > class Tree
15 {
16 public:
17    Tree(); // constructor
18    void insertNode( const NODETYPE & );
19    void preOrderTraversal() const;
20    void inOrderTraversal() const;
21    void postOrderTraversal() const;
22 private:
23    TreeNode< NODETYPE > *rootPtr;
24
25    // utility functions
26    void insertNodeHelper(TreeNode< NODETYPE> **, const NODETYPE & );
27    void preOrderHelper( TreeNode< NODETYPE > * ) const;
28    void inOrderHelper( TreeNode< NODETYPE > * ) const;
29    void postOrderHelper( TreeNode< NODETYPE > * ) const;
30 }; // end class Tree
31
32 // constructor
33 template< typename NODETYPE >
34 Tree< NODETYPE >::Tree()
35 {
36    rootPtr = 0; // indicate tree is initially empty
37 } // end Tree constructor
38
39 // insert node in Tree
40 template< typename NODETYPE >
41 void Tree< NODETYPE >::insertNode( const NODETYPE &value )
42 {
43    insertNodeHelper( &rootPtr, value );
44 } // end function insertNode
45
46 // utility function called by insertNode; receives a pointer
47 // to a pointer so that the function can modify pointer's value
48 template< typename NODETYPE >
49 void Tree< NODETYPE >::insertNodeHelper(
```

```
50    TreeNode< NODETYPE > **ptr, const NODETYPE &value )
51 {
52    // subtree is empty; create new TreeNode containing value
53    if ( *ptr == 0 )
54       *ptr = new TreeNode< NODETYPE >( value );
55    else // subtree is not empty
56    {
57       // data to insert is less than data in current node
58       if ( value < ( *ptr )->data )
59          insertNodeHelper( &( ( *ptr )->leftPtr ), value );
60       else
61       {
62          // data to insert is greater than data in current node
63          if ( value > ( *ptr )->data )
64             insertNodeHelper( &( ( *ptr )->rightPtr ), value );
65          else // duplicate data value ignored
66             cout << value << " dup" << endl;
67       } // end else
68    } // end else
69 } // end function insertNodeHelper
70
71 // begin preorder traversal of Tree
72 template< typename NODETYPE >
73 void Tree< NODETYPE >::preOrderTraversal() const
74 {
75    preOrderHelper( rootPtr );
76 } // end function preOrderTraversal
77
78 // utility function to perform preorder traversal of Tree
79 template< typename NODETYPE >
80 void Tree<NODETYPE>::preOrderHelper(TreeNode<NODETYPE> *ptr ) const
81 {
82    if ( ptr != 0 )
83    {
84       cout << ptr->data << ' '; // process node
85       preOrderHelper( ptr->leftPtr ); // traverse left subtree
86       preOrderHelper( ptr->rightPtr ); // traverse right subtree
87    } // end if
88 } // end function preOrderHelper
89
90 // begin inorder traversal of Tree
91 template< typename NODETYPE >
92 void Tree< NODETYPE >::inOrderTraversal() const
93 {
94    inOrderHelper( rootPtr );
95 } // end function inOrderTraversal
96
97 // utility function to perform inorder traversal of Tree
98 template< typename NODETYPE >
99 void Tree<NODETYPE>::inOrderHelper(TreeNode<NODETYPE> *ptr ) const
100 {
101    if ( ptr != 0 )
102    {
103       inOrderHelper( ptr->leftPtr ); // traverse left subtree
104       cout << ptr->data << ' '; // process node
105       inOrderHelper( ptr->rightPtr ); // traverse right subtree
106    } // end if
107 } // end function inOrderHelper
108
109 // begin postorder traversal of Tree
110 template< typename NODETYPE >
111 void Tree< NODETYPE >::postOrderTraversal() const
```

```
112 {
113    postOrderHelper( rootPtr );
114 } // end function postOrderTraversal
115
116 // utility function to perform postorder traversal of Tree
117 template< typename NODETYPE >
118 void Tree< NODETYPE >::postOrderHelper(
119    TreeNode< NODETYPE > *ptr ) const
120 {
121    if ( ptr != 0 )
122    {
123       postOrderHelper( ptr->leftPtr ); // traverse left subtree
124       postOrderHelper( ptr->rightPtr ); // traverse right subtree
125       cout << ptr->data << ' '; // process node
126    } // end if
127 } // end function postOrderHelper
128
129 #endif
```

شکل 21-21 | تعریف کلاس Tree.

بحث را با تعریف الگوی کلاس **TreeNode** آغاز می‌کنیم (شکل 20-21) که **Tree<NODETYPE>** را بعنوان **friend** اعلان کرده است (خط 13). با اینکار تمام توابع عضو که از کلاس الگوی Tree بدست می‌آیند (شکل 21-21) دوستان متناظر کلاس الگوی **TreeNode** می‌شوند، از اینرو است که می‌توانند به اعضای **private** شی‌های **TreeNode** از آن نوع دسترسی پیدا کنند. بدلیل اینکه از پارامتر **NODETYPE** بعنوان آرگومان **Tree** در اعلان **friend** استفاده شده است، **TreeNode** می‌تواند فقط از طریق یک **Tree** با همان نوع پردازش شود.

```
1  // Fig. 21.22: Fig21_22.cpp
2  // Tree class test program.
3  #include <iostream>
4  using std::cout;
5  using std::cin;
6  using std::fixed;
7
8  #include <iomanip>
9  using std::setprecision;
10
11 #include "Tree.h" // Tree class definition
12
13 int main()
14 {
15    Tree< int > intTree; // create Tree of int values
16    int intValue;
17
18    cout << "Enter 10 integer values:\n";
19
20    // insert 10 integers to intTree
21    for ( int i = 0; i < 10; i++ )
22    {
23       cin >> intValue;
24       intTree.insertNode( intValue );
25    } // end for
26
27    cout << "\nPreorder traversal\n";
```

```
28    intTree.preOrderTraversal();
29
30    cout << "\nInorder traversal\n";
31    intTree.inOrderTraversal();
32
33    cout << "\nPostorder traversal\n";
34    intTree.postOrderTraversal();
35
36    Tree< double > doubleTree; // create Tree of double values
37    double doubleValue;
38
39    cout << fixed << setprecision( 1 )
40       << "\n\n\nEnter 10 double values:\n";
41
42    // insert 10 doubles to doubleTree
43    for ( int j = 0; j < 10; j++ )
44    {
45       cin >> doubleValue;
46       doubleTree.insertNode( doubleValue );
47    } // end for
48
49    cout << "\nPreorder traversal\n";
50    doubleTree.preOrderTraversal();
51
52    cout << "\nInorder traversal\n";
53    doubleTree.inOrderTraversal();
54
55    cout << "\nPostorder traversal\n";
56    doubleTree.postOrderTraversal();
57
58    cout << endl;
59    return 0;
60 } // end main
```

```
Enter 10 integer values:
50 25 75 12 33 67 88 6 13 68

Preorder traversal
50 25 12 6 13 33 75 67 68 88
Inorder traversal
6 12 13 25 33 50 67 68 75 88
Postorder traversal
6 13 12 33 25 68 67 88 75 50

Enter 10 double values:
39.2 16.5 82.7 3.3 65.2 90.8 1.1 4.4 89.5 92.5

Preorder traversal
39.2 16.5 3.3 1.1 4.4 82.7 65.2 90.8 89.5 92.5
Inoreder traversal
1.1 3.3 4.4 16.5 39.2 65.2 82.7 89.5 90.8 92.5
Postorder traversal
1.1 4.4 3.3 16.5 65.2 89.5 92.5 90.8 82.7 39.2
```

شکل 22-21 |ایجاد و پیمایش درخت باینری.

خطوط 32-30 داده **private** از **TreeNode** را اعلان کرده‌اند- مقدار داده گره، و اشاره‌گرهای **leftPtr** (برای زیردرخت گره چپ) و **rightPtr** (برای زیردرخت گره راست). سازنده (خطوط 22-16)

مبادرت به تنظیم مقدار **data** تدارک دیده شده از سوی آرگومان سازنده و تنظیم اشاره‌گرهای **leftPtr** و **rightPtr** با صفر کرده است. تابع عضو **getData** (خطوط 28-25) مقدار **data** را برگشت می‌دهد.

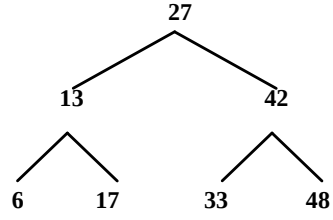
کلاس **Tree** (شکل 21-21) حاوی گره ریشه (**rootPtr**) در خط 22 است که به گره ریشه در درخت اشاره دارد. همچنین کلاس دارای تابع **insertNode** (خطوط 20-17) است که گره‌ای را در درخت جای می‌دهد، البته دارای توابع سراسری **preOrderTraversal**، **inOrderTraversal** و **postOrderTraversal** است که وظیفه پیمایش درخت را برعهده دارند. هر تابع پیمایش درخت، یک تابع یوتیلیتی بازگشتی جداگانه را برای انجام عملیات پیمایش بر روی ساختار داخلی درخت فراخوانی می‌کند. سازنده **Tree** مبادرت به مقداردهی **rootPtr** با **null** (صفر) می‌کند تا نشان دهد درخت در ابتدای کار تهی است.

تابع **InsertNode** ابتدا تابع کمک **insertNodeHelper** (خطوط 68-47) را فراخوانی می‌کند تا بصورت بازگشتی گره‌ای را وارد درخت کند. در یک درخت جستجوی باینری گره‌ها فقط می‌توانند به عنوان گره‌های برگ وارد درخت شوند. اگر درخت تهی باشد، یک **TreeNode** جدید ایجاد، مقداردهی اولیه شده و در درخت درج می‌شود (خطوط 54-53).

اگر درخت تهی نباشد، برنامه مقدار وارد شده را با مقدار **data** در گره ریشه مقایسه می‌کند. اگر مقدار واردهٔ کمتر از (کوچکتر از) از مقدار گره ریشه باشد (خط 57)، برنامه بصورت بازگشتی تابع **insertNodeHelper** (خط 58) را برای درج مقدار در زیر درخت چپ فراخوانی می‌کند. اگر مقدار واردهٔ بزرگتر از مقدار گره ریشه باشد (خط 62)، برنامه بصورت بازگشتی تابع **insertNodeHelper** (خط 64) را برای درج مقدار در زیر درخت راست فراخوانی می‌کند. اگر مقدار وارد شده برابر با مقدار گره ریشه باشد، برنامه پیغام "dup" را چاپ (خط 65) کرده و بدون اینکه مقدار تکراری را وارد درخت کند، باز می‌گردد. دقت کنید که **insertNode** آدرس **rootPtr** را به **insertNodeHelper** ارسال می‌کند (خط 42) از اینرو است که می‌تواند مقدار ذخیره شده در **rootPtr** را اصلاح کند. برای دریافت اشاره‌گر به **rootPtr** که خود یک اشاره‌گر است، اولین آرگومان **insertNodeHelper** بعنوان یک اشاره‌گر به اشاره‌گر به یک **treeNode** اعلان شده است.

توابع پیمایش **inOrderTraversal** (خطوط 94-90)، **preOrderTraversal** (خطوط 75-71) و **postOrderTraversal** (خطوط 113-109) توابع کمکی **inOrderHelper** (خط102)، **preOrderHelper** (خط84) و **postOrderHelper** (خط123) را به ترتیب فراخوانی می‌کنند تا درخت

پیمایش شده و مقادیر هر گره به نمایش در آید. تابعهای کمک کننده در کلاس **Tree** به برنامه‌نویس امکان می‌دهند تا شروع به پیمایش درخت نماید بدون اینکه ابتدا به گره ریشه اشاره کند. برای اینکه بهتر با مبحث پیمایش آشنا شوید از تصویر درخت جستجوی باینری در شکل 23-21 استفاده می‌کنیم.



شکل 23-21 | یک درخت جستجوی باینری.

الگوریتم پیمایش Inorder

تابع **inOrderHelper** تعریف کننده مراحل پیمایش بفرم inorder است. این مراحل عبارتند از:

1- اگر آرگومان برابر **null** باشد، بلافاصله برگشت داده می‌شود.

2- با فراخوانی **inOrderHelper** زیر درخت چپ را پیمایش می‌کند (خط 102).

3- مقدار موجود در گره پردازش می‌شود (خط 103).

4- پیمایش زیر درخت راست با فراخوانی **inOrderHelper** (خط 104).

در پیمایش inorder تا زمانیکه مقادیر گره‌های زیر درخت چپ پردازش نشده، مقدار موجود در گره پردازش نخواهد شد. پیمایش inorder بر روی درخت شکل 23-21 نتیجه زیر را خواهد داشت:

6 13 17 27 33 42 48

دقت کنید که در این نوع پیمایش بر روی درخت جستجوی باینری مقادیر گره‌ها بصورت صعودی بچاپ می‌رسند و اصولاً فرآیند ایجاد یک درخت جستجوی باینری بصورت مرتب شده است.

الگوریتم پیمایش Preorder

تابع **preOrderHelper** تعریف کننده مراحل پیمایش بفرم *preorder* است. این مراحل عبارتند از:

1- اگر آرگومان **null** باشد، بلافاصله برگشت داده می‌شود.

2- پردازش مقدار موجود در گره (خط 83).

3- پیمایش زیر درخت چپ با فراخوانی **preOrderHelper** (خط 84).

4- پیمایش زیر درخت راست با فراخوانی **preOrderHelper** (خط 85).

در پیمایش *preorder* مقدار هر گره ملاقات شده پردازش می‌شود. پس از پردازش مقدار گره بدست آمده، پیمایش کار خود را با پردازش مقادیر در زیر درخت چپ ادامه داده و سپس مقادیر زیر درخت راست پردازش می‌شود. نتیجه پیمایش preorder بر روی درخت شکل 23-21 بصورت زیر خواهد بود:

27 13 6 17 42 33 48

الگوریتم پیمایش Postorder

تابع **postOrderHelper** تعریف کننده مراحل پیمایش بفرم *postorder* است. این مراحل عبارتند از:

1- اگر آرگومان **null** باشد، بلافاصله برگشت داده می‌شود.

2- پیمایش زیر درخت چپ با فراخوانی **postOrderHepler** (خط 122).

3- پیمایش زیر درخت راست با فراخوانی **postOrderHelper** (خط 123).

4- پردازش مقدار موجود در گره (خط 124).

نتیجه پیمایش postroder بر روی درخت شکل 23-21 بصورت زیر خواهد بود:

6 17 13 33 48 42 27

تمرینات

21-1 برنامه‌ای بنویسید که دو لیست پیوندی را به یکدیگر متصل کند.

21-2 برنامه‌ای بنویسید که دو لیست پیوندی مرتب شده با مقادیر صحیح را باهم ادغام کرده و یک لیست مرتب شده ایجاد نماید.

21-3 برنامه‌ای بنویسید که 25 عدد تصادفی از میان اعداد از 0 تا 100 را بصورت مرتب شده در یک لیست پیوندی قرار دهد.

21-4 برنامه‌ای بنویسید که عبارتی دریافت و با استفاده از پشته، آن عبارت را بفرم معکوس به نمایش در آورد.

21-5 برنامه‌ای بنویسید که با استفاده از پشته مشخص کند که آیا رشته‌ای پالندروم است یا خیر.

21-6 روالی بنام depth بنویسید که یک درخت باینری دریافت و عمق آنرا بدست آورد.

21-7 برنامه‌ای بنویسید که یک لیست پیوندی با 10 کاراکتر ایجاد کرده و سپس لیست دیگری که کپی از لیست اولیه است ایجاد کند اما با ترتیب معکوس.

21-8 همانطوری که می‌دانید پشته‌ها توسط کامپایلرها به منظور کمک در پردازش ارزیابی عبارات و ایجاد کد زبان ماشین بکار گرفته می‌شوند. بطور کلی ما انسان‌ها عبارات را بفرم 3+4 یا 7/9 می‌نویسیم که عملگر (+ یا /) در بین عملوندها قرار می‌گیرد، که به اینحالت نشانه‌گذاری میانوندی (infix notation) گفته می‌شود. کامپیوترها حالت نشانه‌گذاری پسوندی (postfix notation) را ترجیح می‌دهند، که در آن عملگر در سمت راست دو عملوند نوشته می‌شود (قرار می‌گیرد). در نشانه‌گذاری پسوندی عبارات 4 + 3 و 7/9 به ترتیب بفرم +34 و /79 نوشته می‌شوند.

برای ارزیابی یک عبارت پیچیده infix، بایستی ابتدا کامپیوتر کل عبارات را بحالت postfix تبدیل کرده و سپس آنرا ارزیابی کند. هر کدام از این الگوریتم‌ها مستلزم یک گذار از چپ به راست بر روی عبارات خواهند داشت.

هر الگوریتم از یک پشته برای انجام عملیات خود استفاده می‌کند و در هر الگوریتم از پشته به منظور متفاوتی استفاده می‌شود. در این تمرین برنامه تبدیل عبارت infix به postfix نوشته خواهد شد و در تمرین بعدی برنامه ارزیابی عبارات postfix نوشته می‌شود.

کلاسی بنام **InfixToPostfixConverter** بنویسید که یک عبارت عادی ریاضی (infix) که فقط متشکل از اعداد صحیح همانند 4 / 8 - 5 * (6+2) است به عبارت postfix تبدیل کند. پس از تبدیل این عبارت، postfix بفرم زیر خواهد بود:

62+5*84/-

برنامه باید عبارت وارد شدهٔ را بدرون رشته **string Builder** خوانده و از کلاس **StackCompostion** برای ایجاد عبارت postfix، در رشته **stringBuilder** استفاده کند. الگوریتم ایجاد عبارت **postfix** بصورت زیر است:

a) پرانتز سمت چپ ')' در درون پشته قرار می‌گیرد.

b) الحاق یک پرانتز راست '(' به انتهای infix.

c) تا زمانیکه پشته خالی نشده، infix از چپ به راست خوانده شده و مراحل زیر انجام می‌شود:

* اگر کاراکتر جاری در infix یک رقم باشد، آنرا به postfix اضافه می‌کند.
* اگر کاراکتر جاری در infix یک پرانتز چپ باشد، آنرا بدرون پشته وارد می‌کند (push).

- اگر کاراکتر جاری در infix یک عملگر باشد:

* خارج کردن عملگرها (اگر عملگری وجود داشته باشد) از بالای پشته (Pop) تا زمانیکه عملگری دارای تقدم برابر یا بالاتر از عملگر فعلی قرار داشته باشد. عملگرهای خارج شده از پشته به postfix الحاق می‌شوند.

* وارد شدن کاراکتر جاری در infix بدرون پشته

- اگر کاراکتر فعلی در infix یک پرانتز سمت راست باشد:

* عملگرها از بالای پشته خارج شده (Pop) و به postfix الحاق می‌شود و اینکار تا زمانی صورت می‌گیرد که یک پرانتز چپ در بالای پشته مشاهده شود.

* پرانتز سمت چپ از پشته خارج می‌شود.

همچنین عملگرهای ریاضی نیز می‌توانند در عبارات وجود داشته باشند، عملگرهای:

+ جمع

- تفریق

* ضرب

/ تقسیم

^ توان

% باقیمانده

تعدادی از تابعهایی که ممکن است در این برنامه از آنها استفاده کنید عبارتنداز:

a) تابع **ConvertToPostfix** برای تبدیل عبارت infix به postfix.

b) تابع **IsOperator** برای تعیین اینکه آیا کاراکتر یک عملگر است یا خیر.

c) تابع **Precedence** برای تعیین اینکه آیا تقدم عملگر **operator1** (از عبارت infix) کمتر، برابر یا بیشتر از تقدم عملگر **operator2** (از پشته) است یا خیر. اگر تقدم عملگر **operator1** کمتر از عملگر **operator2** باشد، تابع مقدار **True** و در غیر اینصورت مقدار **False** برگشت می‌دهد.

9-21 کلاسی با نام **PostfixEvaluator** که عبارات postfix را ارزیابی می‌کند، بنویسید. فرض کنید عبارت postfix بفرم زیر باشد:

62 + 5 * 84 / -

برنامه بایستی عبارت postfix را که متشکل از ارقام و عملگرها می‌باشد بدرون رشته **StringBuilder** وارد کند. برنامه باید عبارت را از ابتدا تا انتها طی کرده و آنرا ارزیابی کند. الگوریتم به شرح زیر عمل می‌کند:

a) الحاق پرانتز سمت راست ')' به انتهای عبارت postfix. هنگامی که با پرانتز سمت راست مواجه شود، ادامه پردازش ضروری نیست.

b) تا زمانیکه با پرانتز سمت راست مواجه نشده است، عبارت از سمت چپ به راست خوانده می‌شود.

- اگر کاراکتر جاری یک رقم باشد:

* دو عنصر از بالای پشته خارج شده (Pop) و در درون متغیرهای **x** و y جای داده می‌شوند.

* محاسبه مقدار، **x عملگر** y.

* وارد کردن نتیجه محاسبه، به درون پشته (Push).

c) هنگامی که با کاراکتر سمت راست مواجه شود، مقدار بالایی پشته از آن خارج می‌شود (Pop) که نتیجه ارزیابی عبارت postfix است.

عملگرهای ریاضی زیر می‌توانند در عبارات وجود داشته باشند:

+ جمع

- تفریق

* ضرب

/ تقسیم

^ توان

% باقیمانده

می‌توانید از تابعهای زیر در برنامه استفاده کنید:

a) تابع **EvaluatePostfixExpression** برای ارزیابی عبارت postfix

b) تابع **Calculate** برای ارزیابی عبارت **Op2** عملگر **Op1**.