



C Functions

C How to Program, 6/e



OBJECTIVES

In this chapter, you'll learn:

- To construct programs modularly from small pieces called functions.
- Common math functions in the C Standard Library.
- To create new functions.
- The mechanisms used to pass information between functions.
- How the function call/return mechanism is supported by the function call stack and activation records.
- Simulation techniques using random number generation.
- How to write and use functions that call themselves.



- 5.1 Introduction
- 5.2 Program Modules in C
- 5.3 Math Library Functions
- 5.4 Functions
- 5.5 Function Definitions
- 5.6 Function Prototypes
- 5.7 Function Call Stack and Activation Records
- 5.8 Headers
- 5.9 Calling Functions By Value and By Reference
- 5.10 Random Number Generation
- 5.11 Example: A Game of Chance
- 5.12 Storage Classes
- 5.13 Scope Rules
- 5.14 Recursion
- 5.15 Example Using Recursion: Fibonacci Series
- 5.16 Recursion vs. Iteration



5.1 Introduction

- ▶ Most computer programs that solve real-world problems are much larger than the programs presented in the first few chapters.
- ▶ Experience has shown that the best way to develop and maintain a large program is to construct it from smaller pieces or **modules**, each of which is more manageable than the original program.
- ▶ This technique is called **divide and conquer**.
- ▶ This chapter describes the features of the C language that facilitate the design, implementation, operation and maintenance of large programs.



5.2 Program Modules in C

- ▶ Modules in C are called **functions**.
- ▶ C programs are typically written by combining new functions you write with “prepackaged” functions available in the **C Standard Library**.
- ▶ The C Standard Library provides a rich collection of functions for performing common mathematical calculations, string manipulations, character manipulations, input/output, and many other useful operations.



Good Programming Practice 5.1

Familiarize yourself with the rich collection of functions in the C Standard Library.



Software Engineering Observation 5.1

Avoid reinventing the wheel. When possible, use C Standard Library functions instead of writing new functions. This can reduce program development time.



Portability Tip 5.1

Using the functions in the C Standard Library helps make programs more portable.



5.2 Program Modules in C (Cont.)

- ▶ The functions `printf`, `scanf` and `pow` that we've used in previous chapters are Standard Library functions.
- ▶ You can write your own functions to define tasks that may be used at many points in a program.
- ▶ These are sometimes referred to as **programmer-defined functions**.
- ▶ The statements defining the function are written only once, and the statements are hidden from other functions.
- ▶ Functions are **invoked** by a **function call**, which specifies the function name and provides information (as **arguments**) that the called function needs to perform its designated task.



5.2 Program Modules in C (Cont.)

- ▶ A common analogy for this is the hierarchical form of management.
- ▶ A boss (the **calling function** or **caller**) asks a worker (the **called function**) to perform a task and report back when the task is done (Fig. 5.1).
- ▶ For example, a function needing to display information on the screen calls the worker function `printf` to perform that task, then `printf` displays the information and reports back—or **returns**—to the calling function when its task is completed.
- ▶ The boss function does not know how the worker function performs its designated tasks.



5.2 Program Modules in C (Cont.)

- ▶ The worker may call other worker functions, and the boss will be unaware of this.
- ▶ We'll soon see how this “hiding” of implementation details promotes good software engineering.
- ▶ Figure 5.1 shows the `main` function communicating with several worker functions in a hierarchical manner.
- ▶ Note that `worker1` acts as a boss function to `worker4` and `worker5`.
- ▶ Relationships among functions may differ from the hierarchical structure shown in this figure.

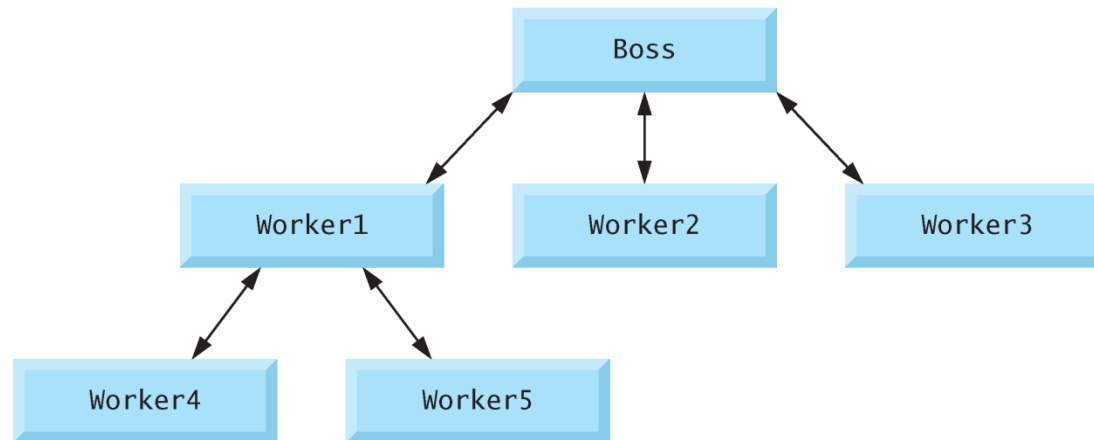


Fig. 5.1 | Hierarchical boss function/worker function relationship.



5.3 Math Library Functions

- ▶ Math library functions allow you to perform certain common mathematical calculations.
- ▶ Functions are normally used in a program by writing the name of the function followed by a left parenthesis followed by the **argument** (or a comma-separated list of arguments) of the function followed by a right parenthesis.
- ▶ For example, a programmer desiring to calculate and print the square root of 900.0 might write

```
printf( "%.2f", sqrt( 900.0 ) );
```
- ▶ When this statement executes, the math library function **sqrt** is called to calculate the square root of the number contained in the parentheses (900.0).



5.3 Math Library Functions (Cont.)

- ▶ The number `900.0` is the argument of the `sqrt` function.
- ▶ The preceding statement would print `30.00`.
- ▶ The `sqrt` function takes an argument of type `double` and returns a result of type `double`.
- ▶ All functions in the math library that return floating point values return the data type `double`.
- ▶ Note that `double` values, like `float` values, can be output using the `%f` conversion specification.



Error-Prevention Tip 5.1

Include the math header by using the preprocessor directive `#include <math.h>` when using functions in the math library.



5.3 Math Library Functions (Cont.)

- ▶ Function arguments may be constants, variables, or expressions.
- ▶ If $c1 = 13.0$, $d = 3.0$ and $f = 4.0$, then the statement

```
printf( "%.2f", sqrt( c1 + d * f ) );
```
- ▶ calculates and prints the square root of $13.0 + 3.0 * 4.0 = 25.0$, namely 5.00 .
- ▶ Some C math library functions are summarized in Fig. 5.2.
- ▶ In the figure, the variables x and y are of type `double`.



Function	Description	Example
<code>sqrt(x)</code>	square root of x	<code>sqrt(900.0)</code> IS 30.0 <code>sqrt(9.0)</code> IS 3.0
<code>exp(x)</code>	exponential function e^x	<code>exp(1.0)</code> IS 2.718282 <code>exp(2.0)</code> IS 7.389056
<code>log(x)</code>	natural logarithm of x (base e)	<code>log(2.718282)</code> IS 1.0 <code>log(7.389056)</code> IS 2.0
<code>log10(x)</code>	logarithm of x (base 10)	<code>log10(1.0)</code> IS 0.0 <code>log10(10.0)</code> IS 1.0 <code>log10(100.0)</code> IS 2.0
<code>fabs(x)</code>	absolute value of x	<code>fabs(13.5)</code> IS 13.5 <code>fabs(0.0)</code> IS 0.0 <code>fabs(-13.5)</code> IS 13.5
<code>ceil(x)</code>	rounds x to the smallest integer not less than x	<code>ceil(9.2)</code> IS 10.0 <code>ceil(-9.8)</code> IS -9.0
<code>floor(x)</code>	rounds x to the largest integer not greater than x	<code>floor(9.2)</code> IS 9.0 <code>floor(-9.8)</code> IS -10.0

Fig. 5.2 | Commonly used math library functions. (Part I of 2.)



Function	Description	Example
<code>pow(x, y)</code>	x raised to power y (x^y)	<code>pow(2, 7)</code> is 128.0 <code>pow(9, .5)</code> is 3.0
<code>fmod(x, y)</code>	remainder of x/y as a floating-point number	<code>fmod(13.657, 2.333)</code> is 1.992
<code>sin(x)</code>	trigonometric sine of x (x in radians)	<code>sin(0.0)</code> is 0.0
<code>cos(x)</code>	trigonometric cosine of x (x in radians)	<code>cos(0.0)</code> is 1.0
<code>tan(x)</code>	trigonometric tangent of x (x in radians)	<code>tan(0.0)</code> is 0.0

Fig. 5.2 | Commonly used math library functions. (Part 2 of 2.)



5.4 Functions

- ▶ Functions allow you to modularize a program.
- ▶ All variables defined in function definitions are **local variables**—they're known only in the function in which they're defined.
- ▶ Most functions have a list of **parameters** that provide the means for communicating information between functions.
- ▶ A function's parameters are also local variables of that function.



Software Engineering Observation 5.2

In programs containing many functions, main is often implemented as a group of calls to functions that perform the bulk of the program's work.



5.4 Functions (Cont.)

- ▶ There are several motivations for “functionalizing” a program.
- ▶ The divide-and-conquer approach makes program development more manageable.
- ▶ Another motivation is **software reusability**—using existing functions as building-blocks to create new programs.
- ▶ Software reusability is a major factor in the object-oriented programming movement that you’ll learn more about when you study languages derived from C, such as C++, Java and C# (pronounced “C sharp”).
- ▶ We use abstraction each time we use standard library functions like **printf**, **scanf** and **pow**.
- ▶ A third motivation is to avoid repeating code in a program.
- ▶ Packaging code as a function allows the code to be executed from several locations in a program simply by calling the function.



Software Engineering Observation 5.3

Each function should be limited to performing a single, well-defined task, and the function name should express that task. This facilitates abstraction and promotes software reusability.



Software Engineering Observation 5.4

If you cannot choose a concise name that expresses what the function does, it's possible that your function is attempting to perform too many diverse tasks. It's usually best to break such a function into several smaller functions—sometimes called decomposition.



5.4 Math Library Functions (Cont.)

- ▶ With good function naming and definition, programs can be created from standardized functions that accomplish specific tasks, rather than being built by using customized code.
- ▶ This is known as **abstraction**.
- ▶ We use abstraction each time we use standard library functions like `printf`, `scanf` and `pow`.
- ▶ A third motivation is to avoid repeating code in a program.
- ▶ Packaging code as a function allows the code to be executed from several locations in a program simply by calling the function.



Software Engineering Observation 5.5

Each function should be limited to performing a single, well-defined task, and the function name should express that task. This facilitates abstraction and promotes software reusability.



Software Engineering Observation 5.6

If you cannot choose a concise name that expresses what the function does, it's possible that your function is attempting to perform too many diverse tasks. It's usually best to break such a function into several smaller functions—sometimes called decomposition.



5.5 Function Definitions

- ▶ Each program we've presented has consisted of a function called `main` that called standard library functions to accomplish its tasks.
- ▶ We now consider how to write custom functions.
- ▶ Consider a program that uses a function `square` to calculate and print the squares of the integers from 1 to 10 (Fig. 5.3).



Good Programming Practice 5.2

Place a blank line between function definitions to separate the functions and enhance program readability.



```
1  /* Fig. 5.3: fig05_03.c
2     Creating and using a programmer-defined function */
3  #include <stdio.h>
4
5  int square( int y ); /* function prototype */
6
7  /* function main begins program execution */
8  int main( void )
9  {
10     int x; /* counter */
11
12     /* loop 10 times and calculate and output square of x each time */
13     for ( x = 1; x <= 10; x++ ) {
14         printf( "%d ", square( x ) ); /* function call */
15     } /* end for */
16
17     printf( "\n" );
18     return 0; /* indicates successful termination */
19 } /* end main */
20
```

Fig. 5.3 | Using a programmer-defined function. (Part I of 2.)



```
21  /* square function definition returns square of parameter */
22  int square( int y ) /* y is a copy of argument to function */
23  {
24      return y * y; /* returns square of y as an int */
25  } /* end function square */
```

1 4 9 16 25 36 49 64 81 100

Fig. 5.3 | Using a programmer-defined function. (Part 2 of 2.)



5.5 Function Definitions (Cont.)

- ▶ Function `square` is **invoked** or **called** in `main` within the `printf` statement (line 14)
`printf( "%d ", square( x ) ); /* function call */`
- ▶ Function `square` receives a copy of the value of `x` in the parameter `y` (line 22).
- ▶ Then `square` calculates `y * y` (line 24).
- ▶ The result is passed back to function `printf` in `main` where `square` was invoked (line 14), and `printf` displays the result.
- ▶ This process is repeated 10 times using the `for` repetition statement.

5.5 Function Definitions (Cont.)

- ▶ The definition of function `square` shows that `square` expects an integer parameter `y`.
- ▶ The keyword `int` preceding the function name (line 22) indicates that `square` returns an integer result.
- ▶ The `return` statement in `square` passes the result of the calculation back to the calling function.
- ▶ Line 5
`int square( int y ); /* function prototype */`
is a `function prototype`.
- ▶ The `int` in parentheses informs the compiler that `square` expects to receive an integer value from the caller.



5.5 Function Definitions (Cont.)

- ▶ The `int` to the left of the function name `square` informs the compiler that `square` returns an integer result to the caller.
- ▶ The compiler refers to the function prototype to check that calls to `square` (line 14) contain the correct return type, the correct number of arguments, the correct argument types, and that the arguments are in the correct order.
- ▶ The format of a function definition is

```
return-value-type function-name ( parameter-list )  
{  
    definitions  
    statements  
}
```



5.5 Function Definitions (Cont.)

- ▶ The *function-name* is any valid identifier.
- ▶ The *return-value-type* is the data type of the result returned to the caller.
- ▶ The *return-value-type* `void` indicates that a function does not return a value.
- ▶ Together, the *return-value-type*, *function-name* and *parameter-list* are sometimes referred to as the function *header*.



Common Programming Error 5.1

Forgetting to return a value from a function that is supposed to return a value can lead to unexpected errors. The C standard states that the result of this omission is undefined.



Common Programming Error 5.2

Returning a value from a function with a `void` return type is a compilation error.



5.5 Function Definitions (Cont.)

- ▶ The *parameter-list* is a comma-separated list that specifies the parameters received by the function when it's called.
- ▶ If a function does not receive any values, *parameter-list* is `void`.
- ▶ A type must be listed explicitly for each parameter.



Common Programming Error 5.3

Specifying function parameters of the same type as double x, y instead of double x, double y results in a compilation error.



Common Programming Error 5.4

Placing a semicolon after the right parenthesis enclosing the parameter list of a function definition is a syntax error.



Common Programming Error 5.5

Defining a parameter again as a local variable in a function is a compilation error.



Good Programming Practice 5.3

Although it's not incorrect to do so, do not use the same names for a function's arguments and the corresponding parameters in the function definition. This helps avoid ambiguity.



5.5 Function Definitions (Cont.)

- ▶ The *definitions* and *statements* within braces form the **function body**.
- ▶ The function body is also referred to as a **block**.
- ▶ Variables can be declared in any block, and blocks can be nested.
- ▶ A function cannot be defined inside another function.



Common Programming Error 5.6

Defining a function inside another function is a syntax error.



Good Programming Practice 5.4

Choosing meaningful function names and meaningful parameter names makes programs more readable and helps avoid excessive use of comments.



Software Engineering Observation 5.7

A function should generally be no longer than one page. Better yet, functions should generally be no longer than half a page. Small functions promote software reusability.



Software Engineering Observation 5.8

Programs should be written as collections of small functions. This makes programs easier to write, debug, maintain and modify.



Software Engineering Observation 5.9

A function requiring a large number of parameters may be performing too many tasks. Consider dividing the function into smaller functions that perform the separate tasks. The function header should fit on one line if possible.



Software Engineering Observation 5.10

The function prototype, function header and function calls should all agree in the number, type, and order of arguments and parameters, and in the type of return value.



5.5 Function Definitions (Cont.)

- ▶ There are three ways to return control from a called function to the point at which a function was invoked.
- ▶ If the function does not return a result, control is returned simply when the function-ending right brace is reached, or by executing the statement
`return;`
- ▶ If the function does return a result, the statement
`return expression;`
- ▶ returns the value of *expression to the caller*.



5.5 Function Definitions (Cont.)

- ▶ Our second example uses a programmer-defined function `maximum` to determine and return the largest of three integers (Fig. 5.4).
- ▶ Three integers are passed to `maximum` (line 19), which determines the largest integer.
- ▶ This value is returned to main by the `return` statement in `maximum` (line 37).



```
1  /* Fig. 5.4: fig05_04.c
2     Finding the maximum of three integers */
3  #include <stdio.h>
4
5  int maximum( int x, int y, int z ); /* function prototype */
6
```

Fig. 5.4 | Finding the maximum of three integers. (Part I of 4.)



```
7  /* function main begins program execution */
8  int main( void )
9  {
10     int number1; /* first integer */
11     int number2; /* second integer */
12     int number3; /* third integer */
13
14     printf( "Enter three integers: " );
15     scanf( "%d%d%d", &number1, &number2, &number3 );
16
17     /* number1, number2 and number3 are arguments
18        to the maximum function call */
19     printf( "Maximum is: %d\n", maximum( number1, number2, number3 ) );
20     return 0; /* indicates successful termination */
21 } /* end main */
22
```

Fig. 5.4 | Finding the maximum of three integers. (Part 2 of 4.)



```
23  /* Function maximum definition */
24  /* x, y and z are parameters */
25  int maximum( int x, int y, int z )
26  {
27      int max = x; /* assume x is largest */
28
29      if ( y > max ) { /* if y is larger than max, assign y to max */
30          max = y;
31      } /* end if */
32
33      if ( z > max ) { /* if z is larger than max, assign z to max */
34          max = z;
35      } /* end if */
36
37      return max; /* max is largest value */
38  } /* end function maximum */
```

Fig. 5.4 | Finding the maximum of three integers. (Part 3 of 4.)



Enter three integers: 22 85 17
Maximum is: 85

Enter three integers: 85 22 17
Maximum is: 85

Enter three integers: 22 17 85
Maximum is: 85

Fig. 5.4 | Finding the maximum of three integers. (Part 4 of 4.)



5.6 Function Prototypes

- ▶ One of the most important features of C is the function prototype.
- ▶ This feature was borrowed by the C standard committee from the developers of C++.
- ▶ A function prototype tells the compiler the type of data returned by the function, the number of parameters the function expects to receive, the types of the parameters, and the order in which these parameters are expected.
- ▶ The compiler uses function prototypes to validate function calls.



5.6 Function Definitions (Cont.)

- ▶ Previous versions of C did not perform this kind of checking, so it was possible to call functions improperly without the compiler detecting the errors.
- ▶ Such calls could result in fatal execution-time errors or nonfatal errors that caused subtle, difficult-to-detect logic errors.



Good Programming Practice 5.5

Include function prototypes for all functions to take advantage of C's type-checking capabilities. Use `#include` preprocessor directives to obtain function prototypes for the standard library functions from the headers for the appropriate libraries, or to obtain headers containing function prototypes for functions developed by you and/or your group members.

5.6 Function Definitions (Cont.)

- ▶ The function prototype for `maximum` in Fig. 5.4 (line 5) is

```
/* function prototype */  
int maximum( int x, int y, int z );
```

- ▶ This function prototype states that `maximum` takes three arguments of type `int` and returns a result of type `int`.
- ▶ Notice that the function prototype is the same as the first line of the function definition of `maximum`.



Good Programming Practice 5.6

Parameter names are sometimes included in function prototypes (our preference) for documentation purposes. The compiler ignores these names.



Common Programming Error 5.7

Forgetting the semicolon at the end of a function prototype is a syntax error.



5.6 Function Definitions (Cont.)

- ▶ A function call that does not match the function prototype is a compilation error.
- ▶ An error is also generated if the function prototype and the function definition disagree.
- ▶ For example, in Fig. 5.4, if the function prototype had been written
`void maximum( int x, int y, int z );`
 - ▶ the compiler would generate an error because the `void` return type in the function prototype would differ from the `int` return type in the function header.



5.6 Function Definitions (Cont.)

- ▶ Another important feature of function prototypes is the **coercion of arguments**, i.e., the forcing of arguments to the appropriate type.
- ▶ For example, the math library function `sqrt` can be called with an integer argument even though the function prototype in `<math.h>` specifies a `double` argument, and the function will still work correctly.
- ▶ The statement

```
printf( "%.3f\n", sqrt( 4 ) );
```

correctly evaluates `sqrt( 4 )`, and prints the value `2.000`.



5.6 Function Definitions (Cont.)

- ▶ The function prototype causes the compiler to convert the integer value `4` to the `double` value `4.0` before the value is passed to `sqrt`.
- ▶ In general, argument values that do not correspond precisely to the parameter types in the function prototype are converted to the proper type before the function is called.
- ▶ These conversions can lead to incorrect results if C's [promotion rules](#) are not followed.
- ▶ The promotion rules specify how types can be converted to other types without losing data.



5.6 Function Definitions (Cont.)

- ▶ In our `sqrt` example above, an `int` is automatically converted to a `double` without changing its value.
- ▶ However, a `double` converted to an `int` truncates the fractional part of the `double` value.
- ▶ Converting large integer types to small integer types (e.g., `long` to `short`) may also result in changed values.
- ▶ The promotion rules automatically apply to expressions containing values of two or more data types (also referred to as **mixed-type expressions**).



5.6 Function Definitions (Cont.)

- ▶ The type of each value in a mixed-type expression is automatically promoted to the “highest” type in the expression (actually a temporary version of each value is created and used for the expression—the original values remain unchanged).
- ▶ Figure 5.5 lists the data types in order from highest type to lowest type with each type’s `printf` and `scanf` conversion specifications.



Data type	printf conversion specification	scanf conversion specification
long double	%Lf	%Lf
double	%f	%lf
float	%f	%f
unsigned long int	%lu	%lu
long int	%ld	%ld
unsigned int	%u	%u
int	%d	%d
unsigned short	%hu	%hu
short	%hd	%hd
char	%c	%c

Fig. 5.5 | Promotion hierarchy for data types.



5.6 Function Definitions (Cont.)

- ▶ Converting values to lower types normally results in an incorrect value.
- ▶ Therefore, a value can be converted to a lower type only by explicitly assigning the value to a variable of lower type, or by using a cast operator.
- ▶ Function argument values are converted to the parameter types in a function prototype as if they were being assigned directly to variables of those types.
- ▶ If our **square** function that uses an integer parameter (Fig. 5.3) is called with a floating-point argument, the argument is converted to **int** (a lower type), and **square** usually returns an incorrect value.
- ▶ For example, **square(4.5)** returns **16**, not **20.25**.



Common Programming Error 5.8

Converting from a higher data type in the promotion hierarchy to a lower type can change the data value. Many compilers issue warnings in such cases.



5.6 Function Definitions (Cont.)

- ▶ If there is no function prototype for a function, the compiler forms its own function prototype using the first occurrence of the function—either the function definition or a call to the function.
- ▶ This typically leads to warnings or errors, depending on the compiler.



Error-Prevention Tip 5.2

Always include function prototypes for the functions you define or use in your program to help prevent compilation errors and warnings.



Software Engineering Observation 5.11

A function prototype placed outside any function definition applies to all calls to the function appearing after the function prototype in the file. A function prototype placed in a function applies only to calls made in that function.



5.7 Function Call Stack and Activation Records

- ▶ To understand how C performs function calls, we first need to consider a data structure (i.e., collection of related data items) known as a **stack**.
- ▶ Students can think of a stack as analogous to a pile of dishes.
- ▶ When a dish is placed on the pile, it's normally placed at the top (referred to as **pushing** the dish onto the stack).
- ▶ Similarly, when a dish is removed from the pile, it's always removed from the top (referred to as **popping** the dish off the stack).
- ▶ Stacks are known as **last-in, first-out (LIFO)** data structures—the last item pushed (inserted) on the stack is the first item popped (removed) from the stack.



5.7 Function Call Stack and Activation Records (Cont.)

- ▶ When a program calls a function, the called function must know how to return to its caller, so the return address of the calling function is pushed onto the **program execution stack** (sometimes referred to as the **function call stack**).
- ▶ If a series of function calls occurs, the successive return addresses are pushed onto the stack in last-in, first-out order so that each function can return to its caller.
- ▶ The program execution stack also contains the memory for the local variables used in each invocation of a function during a program's execution.



5.7 Function Call Stack and Activation Records (Cont.)

- ▶ This data, stored as a portion of the program execution stack, is known as the **activation record** or **stack frame** of the function call.
- ▶ When a function call is made, the activation record for that function call is pushed onto the program execution stack.
- ▶ When the function returns to its caller, the activation record for this function call is popped off the stack and those local variables are no longer known to the program.



5.7 Function Call Stack and Activation Records (Cont.)

- ▶ Of course, the amount of memory in a computer is finite, so only a certain amount of memory can be used to store activation records on the program execution stack.
- ▶ If more function calls occur than can have their activation records stored on the program execution stack, an error known as a **stack overflow** occurs.



5.8 Headers

- ▶ Each standard library has a corresponding **header** containing the function prototypes for all the functions in that library and definitions of various data types and constants needed by those functions.
- ▶ Figure 5.6 lists alphabetically some of the standard library headers that may be included in programs.
- ▶ The term “macros” that is used several times in Fig. 5.6 is discussed in detail in Chapter 13, C Preprocessor.
- ▶ You can create custom headers.
- ▶ Programmer-defined headers should also use the `.h` filename extension.



5.8 Headers (Cont.)

- ▶ A programmer-defined header can be included by using the `#include` preprocessor directive.
- ▶ For example, if the prototype for our square function was located in the header `square.h`, we'd include that header in our program by using the following directive at the top of the program:

```
#include "square.h"
```



Header	Explanation
<code><assert.h></code>	Contains macros and information for adding diagnostics that aid program debugging.
<code><ctype.h></code>	Contains function prototypes for functions that test characters for certain properties, and function prototypes for functions that can be used to convert lowercase letters to uppercase letters and vice versa.
<code><errno.h></code>	Defines macros that are useful for reporting error conditions.
<code><float.h></code>	Contains the floating-point size limits of the system.
<code><limits.h></code>	Contains the integral size limits of the system.
<code><locale.h></code>	Contains function prototypes and other information that enables a program to be modified for the current locale on which it's running. The notion of locale enables the computer system to handle different conventions for expressing data like dates, times, dollar amounts and large numbers throughout the world.
<code><math.h></code>	Contains function prototypes for math library functions.
<code><setjmp.h></code>	Contains function prototypes for functions that allow bypassing of the usual function call and return sequence.

Fig. 5.6 | Some of the standard library headers. (Part I of 2.)



Header	Explanation
<code><signal.h></code>	Contains function prototypes and macros to handle various conditions that may arise during program execution.
<code><stdarg.h></code>	Defines macros for dealing with a list of arguments to a function whose number and types are unknown.
<code><stddef.h></code>	Contains common type definitions used by C for performing calculations.
<code><stdio.h></code>	Contains function prototypes for the standard input/output library functions, and information used by them.
<code><stdlib.h></code>	Contains function prototypes for conversions of numbers to text and text to numbers, memory allocation, random numbers, and other utility functions.
<code><string.h></code>	Contains function prototypes for string-processing functions.
<code><time.h></code>	Contains function prototypes and types for manipulating the time and date.

Fig. 5.6 | Some of the standard library headers. (Part 2 of 2.)



5.9 Calling Functions By Value and By Reference

- ▶ There are two ways to invoke functions in many programming languages—**call-by-value** and **call-by-reference**.
- ▶ When arguments are passed by value, a copy of the argument's value is made and passed to the called function.
- ▶ Changes to the copy do not affect an original variable's value in the caller.
- ▶ When an argument is passed by reference, the caller allows the called function to modify the original variable's value.
- ▶ Call-by-value should be used whenever the called function does not need to modify the value of the caller's original variable.



5.9 Calling Functions By Value and By Reference (Cont.)

- ▶ This prevents the accidental **side effects** (variable modifications) that so greatly hinder the development of correct and reliable software systems.
- ▶ Call-by-reference should be used only with trusted called functions that need to modify the original variable.
- ▶ In C, all calls are by value.
- ▶ In Chapter 6, we'll see that arrays are automatically passed by reference.



5.10 Random Number Generation

- ▶ We now take a brief and, hopefully, entertaining diversion into a popular programming application, namely simulation and game playing.
- ▶ The element of chance can be introduced into computer applications by using the C Standard Library function `rand` from the `<stdlib.h>` header.
- ▶ Consider the following statement:
`i = rand();`
- ▶ The `rand` function generates an integer between 0 and `RAND_MAX` (a symbolic constant defined in the `<stdlib.h>` header).



5.10 Random Number Generation (Cont.)

- ▶ Standard C states that the value of `RAND_MAX` must be at least 32767, which is the maximum value for a two-byte (i.e., 16-bit) integer.
- ▶ The programs in this section were tested on a C system with a maximum value of 32767 for `RAND_MAX`.
- ▶ If `rand` truly produces integers at random, every number between 0 and `RAND_MAX` has an equal chance (or probability) of being chosen each time `rand` is called.
- ▶ The range of values produced directly by `rand` is often different from what is needed in a specific application.



5.10 Random Number Generation (Cont.)

- ▶ For example, a program that simulates coin tossing might require only 0 for “heads” and 1 for “tails.”
- ▶ A dice-rolling program that simulates a six-sided die would require random integers from 1 to 6.
- ▶ To demonstrate `rand`, let’s develop a program to simulate 20 rolls of a six-sided die and print the value of each roll.
- ▶ The function prototype for function `rand` is in `<stdlib.h>`.
- ▶ We use the remainder operator (%) in conjunction with `rand` as follows
 `rand() % 6`
- ▶ to produce integers in the range 0 to 5.



5.10 Random Number Generation (Cont.)

- ▶ This is called **scaling**.
- ▶ The number 6 is called the **scaling factor**.
- ▶ We then **shift** the range of numbers produced by adding 1 to our previous result.
- ▶ The output of Fig. 5.7 confirms that the results are in the range 1 to 6—the output might vary by compiler.



```
1  /* Fig. 5.7: fig05_07.c
2     Shifted, scaled integers produced by 1 + rand() % 6 */
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  /* function main begins program execution */
7  int main( void )
8  {
9     int i; /* counter */
10
11     /* loop 20 times */
12     for ( i = 1; i <= 20; i++ ) {
13
14         /* pick random number from 1 to 6 and output it */
15         printf( "%10d", 1 + ( rand() % 6 ) );
16
17         /* if counter is divisible by 5, begin new line of output */
18         if ( i % 5 == 0 ) {
19             printf( "\n" );
20         } /* end if */
21     } /* end for */
22
23     return 0; /* indicates successful termination */
24 } /* end main */
```

Fig. 5.7 | Shifted, scaled random integers produced by $1 + \text{rand}() \% 6$. (Part I of 2.)



6	6	5	5	6
5	1	1	5	3
6	6	2	4	2
6	2	3	4	1

Fig. 5.7 | Shifted, scaled random integers produced by $1 + \text{rand}() \% 6$. (Part 2 of 2.)



5.10 Random Number Generation (Cont.)

- ▶ To show that these numbers occur approximately with equal likelihood, let's simulate 6000 rolls of a die with the program of Fig. 5.8.
- ▶ Each integer from 1 to 6 should appear approximately 1000 times.
- ▶ As the program output shows, by scaling and shifting we've used the `rand` function to realistically simulate the rolling of a six-sided die.
- ▶ No `default` case is provided in the `switch` statement.



5.10 Random Number Generation (Cont.)

- ▶ Also note the use of the `%S` conversion specifier to print the character strings `"Face"` and `"Frequency"` as column headers (line 53).
- ▶ After we study arrays in Chapter 6, we'll show how to replace this entire `switch` statement elegantly with a single-line statement.



```
1  /* Fig. 5.8: fig05_08.c
2     Roll a six-sided die 6000 times */
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  /* function main begins program execution */
7  int main( void )
8  {
9     int frequency1 = 0; /* rolled 1 counter */
10    int frequency2 = 0; /* rolled 2 counter */
11    int frequency3 = 0; /* rolled 3 counter */
12    int frequency4 = 0; /* rolled 4 counter */
13    int frequency5 = 0; /* rolled 5 counter */
14    int frequency6 = 0; /* rolled 6 counter */
15
16    int roll; /* roll counter, value 1 to 6000 */
17    int face; /* represents one roll of the die, value 1 to 6 */
18
```

Fig. 5.8 | Rolling a six-sided die 6000 times. (Part I of 4.)



```
19  /* loop 6000 times and summarize results */
20  for ( roll = 1; roll <= 6000; roll++ ) {
21      face = 1 + rand() % 6; /* random number from 1 to 6 */
22
23      /* determine face value and increment appropriate counter */
24      switch ( face ) {
25
26          case 1: /* rolled 1 */
27              ++frequency1;
28              break;
29
30          case 2: /* rolled 2 */
31              ++frequency2;
32              break;
33
34          case 3: /* rolled 3 */
35              ++frequency3;
36              break;
37
38          case 4: /* rolled 4 */
39              ++frequency4;
40              break;
41
```

Fig. 5.8 | Rolling a six-sided die 6000 times. (Part 2 of 4.)



```
42         case 5: /* rolled 5 */
43             ++frequency5;
44             break;
45
46         case 6: /* rolled 6 */
47             ++frequency6;
48             break; /* optional */
49     } /* end switch */
50 } /* end for */
51
52 /* display results in tabular format */
53 printf( "%s%13s\n", "Face", "Frequency" );
54 printf( "    1%13d\n", frequency1 );
55 printf( "    2%13d\n", frequency2 );
56 printf( "    3%13d\n", frequency3 );
57 printf( "    4%13d\n", frequency4 );
58 printf( "    5%13d\n", frequency5 );
59 printf( "    6%13d\n", frequency6 );
60 return 0; /* indicates successful termination */
61 } /* end main */
```

Fig. 5.8 | Rolling a six-sided die 6000 times. (Part 3 of 4.)



Face	Frequency
1	1003
2	1017
3	983
4	994
5	1004
6	999

Fig. 5.8 | Rolling a six-sided die 6000 times. (Part 4 of 4.)



5.10 Random Number Generation (Cont.)

- ▶ Executing the program of Fig. 5.7 again produces exactly the same sequence of values.
- ▶ How can these be random numbers? Ironically, this repeatability is an important characteristic of function `rand`.



5.10 Random Number Generation (Cont.)

- ▶ When debugging a program, this repeatability is essential for proving that corrections to a program work properly.
- ▶ Function `rand` actually generates **pseudorandom numbers**.
- ▶ Calling `rand` repeatedly produces a sequence of numbers that appears to be random.
- ▶ However, the sequence repeats itself each time the program is executed.
- ▶ Once a program has been thoroughly debugged, it can be conditioned to produce a different sequence of random numbers for each execution.



5.10 Random Number Generation (Cont.)

- ▶ This is called **randomizing** and is accomplished with the standard library function **srand**.
- ▶ Function **srand** takes an **unsigned** integer argument and **seeds** function **rand** to produce a different sequence of random numbers for each execution of the program.
- ▶ We demonstrate **srand** in Fig. 5.9.



5.10 Random Number Generation (Cont.)

- ▶ A variable of type `unsigned` is also stored in at least two bytes of memory.
- ▶ A two-byte `unsigned int` can have only positive values in the range 0 to 65535.
- ▶ A four-byte `unsigned int` can have only positive values in the range 0 to 4294967295.
- ▶ Function `srand` takes an `unsigned` value as an argument.
- ▶ The conversion specifier `%u` is used to read an `unsigned` value with `scanf`.
- ▶ The function prototype for `srand` is found in `<stdlib.h>`.



```
1  /* Fig. 5.9: fig05_09.c
2     Randomizing die-rolling program */
3  #include <stdlib.h>
4  #include <stdio.h>
5
6  /* function main begins program execution */
7  int main( void )
8  {
9     int i; /* counter */
10    unsigned seed; /* number used to seed random number generator */
11
12    printf( "Enter seed: " );
13    scanf( "%u", &seed ); /* note %u for unsigned */
14
15    srand( seed ); /* seed random number generator */
16
17    /* loop 10 times */
18    for ( i = 1; i <= 10; i++ ) {
19
20        /* pick a random number from 1 to 6 and output it */
21        printf( "%10d", 1 + ( rand() % 6 ) );
22
```

Fig. 5.9 | Randomizing the die-rolling program. (Part I of 2.)



```
23      /* if counter is divisible by 5, begin a new line of output */
24      if ( i % 5 == 0 ) {
25          printf( "\n" );
26      } /* end if */
27  } /* end for */
28
29      return 0; /* indicates successful termination */
30  } /* end main */
```

Enter seed: **67**

6	1	4	6	2
1	6	1	6	4

Enter seed: **867**

2	4	6	1	6
1	1	3	6	2

Enter seed: **67**

6	1	4	6	2
1	6	1	6	4

Fig. 5.9 | Randomizing the die-rolling program. (Part 2 of 2.)



5.10 Random Number Generation (Cont.)

- ▶ Let's run the program several times and observe the results.
- ▶ Notice that a different sequence of random numbers is obtained each time the program is run, provided that a different seed is supplied.
- ▶ To randomize without entering a seed each time, use a statement like

```
srand( time( NULL ) );
```
- ▶ This causes the computer to read its clock to obtain the value for the seed automatically.
- ▶ Function `time` returns the number of seconds that have passed since midnight on January 1, 1970.



5.10 Random Number Generation (Cont.)

- ▶ This value is converted to an unsigned integer and used as the seed to the random number generator.
- ▶ Function `time` takes `NULL` as an argument (`time` is capable of providing you with a string representing the value it returns; `NULL` disables this capability for a specific call to `time`).
- ▶ The function prototype for `time` is in `<time.h>`.



5.10 Random Number Generation (Cont.)

- ▶ The values produced directly by `rand` are always in the range:
$$0 \leq \text{rand}() \leq \text{RAND_MAX}$$
- ▶ As you know, the following statement simulates rolling a six-sided die:
$$\text{face} = 1 + \text{rand}() \% 6;$$
- ▶ This statement always assigns an integer value (at random) to the variable `face` in the range $1 \leq \text{face} \leq 6$.
- ▶ The width of this range (i.e., the number of consecutive integers in the range) is 6 and the starting number in the range is 1.



5.10 Random Number Generation (Cont.)

- ▶ Referring to the preceding statement, we see that the width of the range is determined by the number used to scale `rand` with the remainder operator (i.e., 6), and the starting number of the range is equal to the number (i.e., 1) that is added to `rand % 6`.
- ▶ We can generalize this result as follows
$$n = a + \text{rand}() \% b;$$
- ▶ where `a` is the **shifting value** (which is equal to the first number in the desired range of consecutive integers) and `b` is the scaling factor (which is equal to the width of the desired range of consecutive integers).



Common Programming Error 5.9

Using `srand` in place of `rand` to generate random numbers.



5.11 Example: A Game of Chance

- ▶ One of the most popular games of chance is a dice game known as “craps.” The rules of the game are simple.
 - A player rolls two dice. Each die has six faces. These faces contain 1, 2, 3, 4, 5, and 6 spots. After the dice have come to rest, the sum of the spots on the two upward faces is calculated. If the sum is 7 or 11 on the first throw, the player wins. If the sum is 2, 3, or 12 on the first throw (called “craps”), the player loses (i.e., the “house” wins). If the sum is 4, 5, 6, 8, 9, or 10 on the first throw, then that sum becomes the player’s “point.” To win, you must continue rolling the dice until you “make your point.” The player loses by rolling a 7 before making the point.
- ▶ Figure 5.10 simulates the game of craps and Fig. 5.11 shows several sample executions.



```
1  /* Fig. 5.10: fig05_10.c
2     Craps */
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <time.h> /* contains prototype for function time */
6
7  /* enumeration constants represent game status */
8  enum Status { CONTINUE, WON, LOST };
9
10 int rollDice( void ); /* function prototype */
11
12 /* function main begins program execution */
13 int main( void )
14 {
15     int sum; /* sum of rolled dice */
16     int myPoint; /* point earned */
17
18     enum Status gameStatus; /* can contain CONTINUE, WON, or LOST */
19
20     /* randomize random number generator using current time */
21     srand( time( NULL ) );
22
23     sum = rollDice(); /* first roll of the dice */
```

Fig. 5.10 | Program to simulate the game of craps. (Part I of 4.)



```
24
25  /* determine game status based on sum of dice */
26  switch( sum ) {
27
28      /* win on first roll */
29      case 7:
30      case 11:
31          gameStatus = WON;
32          break;
33
34      /* lose on first roll */
35      case 2:
36      case 3:
37      case 12:
38          gameStatus = LOST;
39          break;
40
41      /* remember point */
42      default:
43          gameStatus = CONTINUE;
44          myPoint = sum;
45          printf( "Point is %d\n", myPoint );
46          break; /* optional */
47  } /* end switch */
```

Fig. 5.10 | Program to simulate the game of craps. (Part 2 of 4.)



```
48
49  /* while game not complete */
50  while ( gameStatus == CONTINUE ) {
51      sum = rollDice(); /* roll dice again */
52
53      /* determine game status */
54      if ( sum == myPoint ) { /* win by making point */
55          gameStatus = WON; /* game over, player won */
56      } /* end if */
57      else {
58          if ( sum == 7 ) { /* lose by rolling 7 */
59              gameStatus = LOST; /* game over, player lost */
60          } /* end if */
61      } /* end else */
62  } /* end while */
63
64  /* display won or lost message */
65  if ( gameStatus == WON ) { /* did player win? */
66      printf( "Player wins\n" );
67  } /* end if */
68  else { /* player lost */
69      printf( "Player loses\n" );
70  } /* end else */
```

Fig. 5.10 | Program to simulate the game of craps. (Part 3 of 4.)



```
71
72     return 0; /* indicates successful termination */
73 } /* end main */
74
75 /* roll dice, calculate sum and display results */
76 int rollDice( void )
77 {
78     int die1; /* first die */
79     int die2; /* second die */
80     int workSum; /* sum of dice */
81
82     die1 = 1 + ( rand() % 6 ); /* pick random die1 value */
83     die2 = 1 + ( rand() % 6 ); /* pick random die2 value */
84     workSum = die1 + die2; /* sum die1 and die2 */
85
86     /* display results of this roll */
87     printf( "Player rolled %d + %d = %d\n", die1, die2, workSum );
88     return workSum; /* return sum of dice */
89 } /* end function rollDice */
```

Fig. 5.10 | Program to simulate the game of craps. (Part 4 of 4.)



Player rolled $5 + 6 = 11$
Player wins

Player rolled $4 + 1 = 5$
Point is 5
Player rolled $6 + 2 = 8$
Player rolled $2 + 1 = 3$
Player rolled $3 + 2 = 5$
Player wins

Player rolled $1 + 1 = 2$
Player loses

Player rolled $6 + 4 = 10$
Point is 10
Player rolled $3 + 4 = 7$
Player loses

Fig. 5.11 | Sample runs for the game of craps.



5.11 Example: A Game of Chance (Conf.)

- ▶ In the rules of the game, notice that the player must roll two dice on the first roll, and must do so later on all subsequent rolls.
- ▶ We define a function `rollDice` to roll the dice and compute and print their sum.
- ▶ Function `rollDice` is defined once, but it's called from two places in the program (lines 23 and 51).
- ▶ Interestingly, `rollDice` takes no arguments, so we've indicated `void` in the parameter list (line 76).
- ▶ Function `rollDice` does return the sum of the two dice, so a return type of `int` is indicated in the function header.



5.11 Example: A Game of Chance (Conf.)

- ▶ The player may win or lose on the first roll, or may win or lose on any subsequent roll.
- ▶ Variable `gameStatus`, defined to be of a new type—`enum Status`—stores the current status.
- ▶ Line 8 creates a programmer-defined type called an **enumeration**.
- ▶ An enumeration, introduced by the keyword `enum`, is a set of integer constants represented by identifiers.
- ▶ **Enumeration constants** are sometimes called symbolic constants.
- ▶ Values in an `enum` start with 0 and are incremented by 1.



5.11 Example: A Game of Chance (Conf.)

- ▶ In line 8, the constant **CONTINUE** has the value 0, **WON** has the value 1 and **LOST** has the value 2.
- ▶ It's also possible to assign an integer value to each identifier in an **enum** (see Chapter 10).
- ▶ The identifiers in an enumeration must be unique, but the values may be duplicated.



Common Programming Error 5.10

Assigning a value to an enumeration constant after it has been defined is a syntax error.



Common Programming Error 5.11

Use only uppercase letters in the names of enumeration constants to make these constants stand out in a program and to indicate that enumeration constants are not variables.

5.11 Example: A Game of Chance (Conf.)

- ▶ When the game is won, either on the first roll or on a subsequent roll, `gameStatus` is set to `WON`.
- ▶ When the game is lost, either on the first roll or on a subsequent roll, `gameStatus` is set to `LOST`.
- ▶ Otherwise `gameStatus` is set to `CONTINUE` and the game continues.
- ▶ After the first roll, if the game is over, the `while` statement (line 50) is skipped because `gameStatus` is not `CONTINUE`.
- ▶ The program proceeds to the `if...else` statement at line 65, which prints "Player wins" if `gameStatus` is `WON` and "Player loses" otherwise.



5.11 Example: A Game of Chance (Conf.)

- ▶ After the first roll, if the game is not over, then `sum` is saved in `myPoint`.
- ▶ Execution proceeds with the `while` statement (line 50) because `gameStatus` is `CONTINUE`.
- ▶ Each time through the `while`, `rollDice` is called to produce a new `sum`.
- ▶ If `sum` matches `myPoint`, `gameStatus` is set to `WON` to indicate that the player won, the `while`-test fails, the `if...else` statement (line 65) prints "`Player wins`" and execution terminates.



5.11 Example: A Game of Chance (Conf.)

- ▶ If `sum` is equal to 7 (line 58), `gameStatus` is set to `LOST` to indicate that the player lost, the `while`-test fails, the `if...else` statement (line 65) prints "`Player loses`" and execution terminates.
- ▶ Note the program's interesting control architecture.
- ▶ We've used two functions—`main` and `rollDice`—and the `switch`, `while`, nested `if...else` and nested `if` statements.
- ▶ In the exercises, we'll investigate various interesting characteristics of the game of craps.



5.12 Storage Classes

- ▶ In Chapters 2–4, we used identifiers for variable names.
- ▶ The attributes of variables include name, type, size and value.
- ▶ In this chapter, we also use identifiers as names for user-defined functions.
- ▶ Actually, each identifier in a program has other attributes, including **storage class**, **storage duration**, **scope** and **linkage**.
- ▶ C provides four storage classes, indicated by the **storage class specifiers**: **auto**, **register**, **extern** and **static**.
- ▶ An identifier's **storage class** determines its storage duration, scope and linkage.
- ▶ An identifier's **storage duration** is the period during which the identifier exists in memory.



5.12 Storage Classes (Cont.)

- ▶ Some exist briefly, some are repeatedly created and destroyed, and others exist for the entire execution of a program.
- ▶ An identifier's **scope** is where the identifier can be referenced in a program.
- ▶ Some can be referenced throughout a program, others from only portions of a program.
- ▶ An identifier's **linkage** determines for a multiple-source-file program (a topic we'll investigate in Chapter 14) whether the identifier is known only in the current source file or in any source file with proper declarations.
- ▶ This section discusses storage classes and storage duration.



5.12 Storage Classes (Cont.)

- ▶ Section 5.13 discusses scope.
- ▶ Chapter 14 discusses identifier linkage and programming with multiple source files.
- ▶ The four storage-class specifiers can be split into two storage durations: **automatic storage duration** and **static storage duration**.
- ▶ Keywords **auto** and **register** are used to declare variables of automatic storage duration.
- ▶ Variables with automatic storage duration are created when the block in which they're defined is entered; they exist while the block is active, and they're destroyed when the block is exited.



5.12 Storage Classes (Cont.)

- ▶ Only variables can have automatic storage duration.
- ▶ A function's local variables (those declared in the parameter list or function body) normally have automatic storage duration.
- ▶ Keyword **auto** explicitly declares variables of automatic storage duration.



5.12 Storage Classes (Cont.)

- ▶ For example, the following declaration indicates that `double` variables `x` and `y` are automatic local variables and they exist only in the body of the function in which the declaration appears:

```
auto double x, y;
```

- ▶ Local variables have automatic storage duration by default, so keyword `auto` is rarely used.
- ▶ For the remainder of the text, we'll refer to variables with automatic storage duration simply as `automatic variables`.



Performance Tip 5.1

Automatic storage is a means of conserving memory, because automatic variables exist only when they're needed. They're created when a function is entered and destroyed when the function is exited.



Software Engineering Observation 5.12

*Automatic storage is an example of the **principle of least privilege**—allowing access to data only when it's absolutely needed. Why have variables stored in memory and accessible when in fact they're not needed?*



5.12 Storage Classes (Cont.)

- ▶ Data in the machine-language version of a program is normally loaded into registers for calculations and other processing.



Performance Tip 5.2

The storage-class specifier `register` can be placed before an automatic variable declaration to suggest that the compiler maintain the variable in one of the computer's high-speed hardware registers. If intensely used variables such as counters or totals can be maintained in hardware registers, the overhead of repeatedly loading the variables from memory into the registers and storing the results back into memory can be eliminated.



5.12 Storage Classes (Cont.)

- ▶ The compiler may ignore `register` declarations.
- ▶ For example, there may not be a sufficient number of registers available for the compiler to use.
- ▶ The following declaration suggests that the integer variable `counter` be placed in one of the computer's registers and initialized to 1:
 - `register int counter = 1;`
- ▶ Keyword `register` can be used only with variables of automatic storage duration.



Performance Tip 5.3

*Often, register declarations are unnecessary. Today's optimizing compilers are capable of recognizing frequently used variables and can decide to place them in registers without the need for a **register** declaration.*



5.12 Storage Classes (Cont.)

- ▶ Keywords **extern** and **static** are used in the declarations of identifiers for variables and functions of static storage duration.
- ▶ Identifiers of static storage duration exist from the time at which the program begins execution.
- ▶ For static variables, storage is allocated and initialized once, when the program begins execution.
- ▶ For functions, the name of the function exists when the program begins execution.



5.12 Storage Classes (Cont.)

- ▶ However, even though the variables and the function names exist from the start of program execution, this does not mean that these identifiers can be accessed throughout the program.
- ▶ Storage duration and scope (where a name can be used) are separate issues, as we'll see in Section 5.13.
- ▶ There are two types of identifiers with static storage duration: external identifiers (such as global variables and function names) and local variables declared with the storage-class specifier `static`.
- ▶ Global variables and function names are of storage class `extern` by default.



5.12 Storage Classes (Cont.)

- ▶ Global variables are created by placing variable declarations outside any function definition, and they retain their values throughout the execution of the program.
- ▶ Global variables and functions can be referenced by any function that follows their declarations or definitions in the file.
- ▶ This is one reason for using function prototypes—when we include `stdio.h` in a program that calls `printf`, the function prototype is placed at the start of our file to make the name `printf` known to the rest of the file.



Software Engineering Observation 5.13

Defining a variable as global rather than local allows unintended side effects to occur when a function that does not need access to the variable accidentally or maliciously modifies it. In general, use of global variables should be avoided except in certain situations with unique performance requirements (as discussed in Chapter 14).



Software Engineering Observation 5.14

Variables used only in a particular function should be defined as local variables in that function rather than as external variables.



5.12 Storage Classes (Cont.)

- ▶ Local variables declared with the keyword `static` are still known only in the function in which they're defined, but unlike automatic variables, `static` local variables retain their value when the function is exited.
- ▶ The next time the function is called, the `static` local variable contains the value it had when the function last exited.
- ▶ The following statement declares local variable `count` to be `static` and to be initialized to 1.
 - `static int count = 1;`



5.12 Storage Classes (Cont.)

- ▶ All numeric variables of static storage duration are initialized to zero if you do not explicitly initialize them.
- ▶ Keywords **extern** and **static** have special meaning when explicitly applied to external identifiers.
- ▶ In Chapter 14 we discuss the explicit use of **extern** and **static** with external identifiers and multiple-source-file programs.



5.13 Scope Rules

- ▶ The **scope of an identifier** is the portion of the program in which the identifier can be referenced.
- ▶ For example, when we define a local variable in a block, it can be referenced only following its definition in that block or in blocks nested within that block.
- ▶ The four identifier scopes are **function scope**, **file scope**, **block scope**, and **function-prototype scope**.
- ▶ Labels (an identifier followed by a colon such as **start:**) are the only identifiers with **function scope**.
- ▶ Labels can be used anywhere in the function in which they appear, but cannot be referenced outside the function body.



5.13 Scope Rules (Cont.)

- ▶ Labels are used in `switch` statements (as `case` labels) and in `goto` statements (see Chapter 14).
- ▶ Labels are implementation details that functions hide from one another.
- ▶ This hiding—more formally called [information hiding](#)—is a means of implementing the [principle of least privilege](#), one of the most fundamental principles of good software engineering.
- ▶ An identifier declared outside any function has [file scope](#).
- ▶ Such an identifier is “known” (i.e., accessible) in all functions from the point at which the identifier is declared until the end of the file.



5.13 Scope Rules (Cont.)

- ▶ Global variables, function definitions, and function prototypes placed outside a function all have file scope.
- ▶ Identifiers defined inside a block have **block scope**.
- ▶ Block scope ends at the terminating right brace (}) of the block.
- ▶ Local variables defined at the beginning of a function have block scope as do function parameters, which are considered local variables by the function.
- ▶ Any block may contain variable definitions.



5.13 Scope Rules (Cont.)

- ▶ When blocks are nested, and an identifier in an outer block has the same name as an identifier in an inner block, the identifier in the outer block is “hidden” until the inner block terminates.
- ▶ This means that while executing in the inner block, the inner block sees the value of its own local identifier and not the value of the identically named identifier in the enclosing block.
- ▶ Local variables declared `static` still have block scope, even though they exist from the time the program begins execution.



5.13 Scope Rules (Cont.)

- ▶ Thus, storage duration does not affect the scope of an identifier.
- ▶ The only identifiers with **function-prototype scope** are those used in the parameter list of a function prototype.
- ▶ As mentioned previously, function prototypes do not require names in the parameter list—only types are required.
- ▶ If a name is used in the parameter list of a function prototype, the compiler ignores the name.
- ▶ Identifiers used in a function prototype can be reused elsewhere in the program without ambiguity.



Common Programming Error 5.12

Accidentally using the same name for an identifier in an inner block as is used for an identifier in an outer block, when in fact you want the identifier in the outer block to be active for the duration of the inner block.



Error-Prevention Tip 5.3

*Avoid variable names that hide names in outer scopes.
This can be accomplished simply by avoiding the use of
duplicate identifiers in a program.*



5.13 Scope Rules (Cont.)

- ▶ Figure 5.12 demonstrates scoping issues with global variables, automatic local variables, and `static` local variables.
- ▶ A global variable `x` is defined and initialized to 1 (line 9).
- ▶ This global variable is hidden in any block (or function) in which a variable named `x` is defined.
- ▶ In `main`, a local variable `x` is defined and initialized to 5 (line 14).
- ▶ This variable is then printed to show that the global `x` is hidden in `main`.
- ▶ Next, a new block is defined in `main` with another local variable `x` initialized to 7 (line 19).



5.13 Scope Rules (Cont.)

- ▶ This variable is printed to show that it hides `x` in the outer block of `main`.
- ▶ The variable `x` with value 7 is automatically destroyed when the block is exited, and the local variable `x` in the outer block of `main` is printed again to show that it's no longer hidden.
- ▶ The program defines three functions that each take no arguments and return nothing.
- ▶ Function `useLocal` defines an automatic variable `x` and initializes it to 25 (line 40).
- ▶ When `useLocal` is called, the variable is printed, incremented, and printed again before exiting the function.



5.13 Scope Rules (Cont.)

- ▶ Each time this function is called, automatic variable `x` is reinitialized to 25.
- ▶ Function `useStaticLocal` defines a `static` variable `x` and initializes it to 50 (line 53).
- ▶ Local variables declared as `static` retain their values even when they're out of scope.
- ▶ When `useStaticLocal` is called, `x` is printed, incremented, and printed again before exiting the function.
- ▶ In the next call to this function, `static` local variable `x` will contain the value 51.
- ▶ Function `useGlobal` does not define any variables.



5.13 Scope Rules (Cont.)

- ▶ Therefore, when it refers to variable `x`, the global `x` (line 9) is used.
- ▶ When `useGlobal` is called, the global variable is printed, multiplied by 10, and printed again before exiting the function.
- ▶ The next time function `useGlobal` is called, the global variable still has its modified value, 10.
- ▶ Finally, the program prints the local variable `x` in `main` again (line 33) to show that none of the function calls modified the value of `x` because the functions all referred to variables in other scopes.



```
1  /* Fig. 5.12: fig05_12.c
2     A scoping example */
3  #include <stdio.h>
4
5  void useLocal( void ); /* function prototype */
6  void useStaticLocal( void ); /* function prototype */
7  void useGlobal( void ); /* function prototype */
8
9  int x = 1; /* global variable */
10
11 /* function main begins program execution */
12 int main( void )
13 {
14     int x = 5; /* local variable to main */
15
16     printf("local x in outer scope of main is %d\n", x );
17
18     { /* start new scope */
19         int x = 7; /* local variable to new scope */
20
21         printf( "local x in inner scope of main is %d\n", x );
22     } /* end new scope */
23
```

Fig. 5.12 | Scoping example. (Part I of 4.)



```
24     printf( "local x in outer scope of main is %d\n", x );
25
26     useLocal(); /* useLocal has automatic local x */
27     useStaticLocal(); /* useStaticLocal has static local x */
28     useGlobal(); /* useGlobal uses global x */
29     useLocal(); /* useLocal reinitializes automatic local x */
30     useStaticLocal(); /* static local x retains its prior value */
31     useGlobal(); /* global x also retains its value */
32
33     printf( "\nlocal x in main is %d\n", x );
34     return 0; /* indicates successful termination */
35 } /* end main */
36
37 /* useLocal reinitializes local variable x during each call */
38 void useLocal( void )
39 {
40     int x = 25; /* initialized each time useLocal is called */
41
42     printf( "\nlocal x in useLocal is %d after entering useLocal\n", x );
43     x++;
44     printf( "local x in useLocal is %d before exiting useLocal\n", x );
45 } /* end function useLocal */
46
```

Fig. 5.12 | Scoping example. (Part 2 of 4.)



```
47  /* useStaticLocal initializes static local variable x only the first time
48     the function is called; value of x is saved between calls to this
49     function */
50  void useStaticLocal( void )
51  {
52     /* initialized only first time useStaticLocal is called */
53     static int x = 50;
54
55     printf( "\nlocal static x is %d on entering useStaticLocal\n", x );
56     x++;
57     printf( "local static x is %d on exiting useStaticLocal\n", x );
58 } /* end function useStaticLocal */
59
60 /* function useGlobal modifies global variable x during each call */
61 void useGlobal( void )
62 {
63     printf( "\nglobal x is %d on entering useGlobal\n", x );
64     x *= 10;
65     printf( "global x is %d on exiting useGlobal\n", x );
66 } /* end function useGlobal */
```

Fig. 5.12 | Scoping example. (Part 3 of 4.)



```
local x in outer scope of main is 5
local x in inner scope of main is 7
local x in outer scope of main is 5

local x in useLocal is 25 after entering useLocal
local x in useLocal is 26 before 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 in useLocal is 25 after entering useLocal
local x in useLocal is 26 before exiting useLocal

local static x is 51 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
```

Fig. 5.12 | Scoping example. (Part 4 of 4.)



5.14 Recursion

- ▶ The programs we've discussed are generally structured as functions that call one another in a disciplined, hierarchical manner.
- ▶ For some types of problems, it's useful to have functions call themselves.
- ▶ A **recursive function** is a function that calls itself either directly or indirectly through another function.
- ▶ Recursion is a complex topic discussed at length in upper-level computer science courses.
- ▶ In this section and the next, simple examples of recursion are presented.



5.14 Recursion (Cont.)

- ▶ This book contains an extensive treatment of recursion, which is spread throughout Chapters 5–8, 12 and Appendix F.
- ▶ Figure 5.17, in Section 5.16, summarizes the 31 recursion examples and exercises in the book.
- ▶ We consider recursion conceptually first, and then examine several programs containing recursive functions.
- ▶ Recursive problem-solving approaches have a number of elements in common.
- ▶ A recursive function is called to solve a problem.
- ▶ The function actually knows how to solve only the simplest case(s), or so-called **base case(s)**.



5.14 Recursion (Cont.)

- ▶ If the function is called with a base case, the function simply returns a result.
- ▶ If the function is called with a more complex problem, the function divides the problem into two conceptual pieces: a piece that the function knows how to do and a piece that it does not know how to do.
- ▶ To make recursion feasible, the latter piece must resemble the original problem, but be a slightly simpler or slightly smaller version.



5.14 Recursion (Cont.)

- ▶ Because this new problem looks like the original problem, the function launches (calls) a fresh copy of itself to go to work on the smaller problem—this is referred to as a **recursive call** and is also called the **recursion step**.
- ▶ The recursion step also includes the keyword **return**, because its result will be combined with the portion of the problem the function knew how to solve to form a result that will be passed back to the original caller, possibly **main**.
- ▶ The recursion step executes while the original call to the function is still open, i.e., it has not yet finished executing.



5.14 Recursion (Cont.)

- ▶ The recursion step can result in many more such recursive calls, as the function keeps dividing each problem it's called with into two conceptual pieces.
- ▶ In order for the recursion to terminate, each time the function calls itself with a slightly simpler version of the original problem, this sequence of smaller problems must eventually converge on the base case.
- ▶ At that point, the function recognizes the base case, returns a result to the previous copy of the function, and a sequence of returns ensues all the way up the line until the original call of the function eventually returns the final result to `main`.

5.14 Recursion (Cont.)

- ▶ The factorial of a nonnegative integer n , written $n!$ (and pronounced “ n factorial”), is the product
 - $n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 1$with $1!$ equal to 1, and $0!$ defined to be 1.
- ▶ For example, $5!$ is the product $5 * 4 * 3 * 2 * 1$, which is equal to 120.
- ▶ The factorial of an integer, **number**, greater than or equal to 0 can be calculated iteratively (nonrecursively) using a **for** statement as follows:

```
factorial = 1;  
for ( counter = number; counter >= 1; counter-- )  
    factorial *= counter;
```



5.14 Recursion (Cont.)

- ▶ A recursive definition of the factorial function is arrived at by observing the following relationship:

$$n! = n \cdot (n - 1)!$$

- ▶ For example, 5! is clearly equal to 5 * 4! as is shown by the following:

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$

$$5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$$

$$5! = 5 \cdot (4!)$$

- ▶ The evaluation of 5! would proceed as shown in Fig. 5.13.



5.14 Recursion (Cont.)

- ▶ Figure 5.13(a) shows how the succession of recursive calls proceeds until $1!$ is evaluated to be 1, which terminates the recursion.
- ▶ Figure 5.13(b) shows the values returned from each recursive call to its caller until the final value is calculated and returned.
- ▶ Figure 5.14 uses recursion to calculate and print the factorials of the integers 0–10 (the choice of the type `long` will be explained momentarily).
- ▶ The recursive `factorial` function first tests whether a terminating condition is true, i.e., whether `number` is less than or equal to 1.



5.14 Recursion (Cont.)

- ▶ If `number` is indeed less than or equal to 1, `factorial` returns 1, no further recursion is necessary, and the program terminates.
- ▶ If `number` is greater than 1, the statement
`return number * factorial( number - 1 );`
- ▶ expresses the problem as the product of `number` and a recursive call to `factorial` evaluating the factorial of `number - 1`.
- ▶ The call `factorial( number - 1 )` is a slightly simpler problem than the original calculation `factorial( number )`.



5.14 Recursion (Cont.)

- ▶ Function `factorial` (line 22) has been declared to receive a parameter of type `long` and return a result of type `long`.
- ▶ This is shorthand notation for `long int`.
- ▶ The C standard specifies that a variable of type `long int` is stored in at least 4 bytes, and thus may hold a value as large as +2147483647.
- ▶ As can be seen in Fig. 5.14, factorial values become large quickly.
- ▶ We've chosen the data type `long` so the program can calculate factorials greater than 7! on computers with small (such as 2-byte) integers.

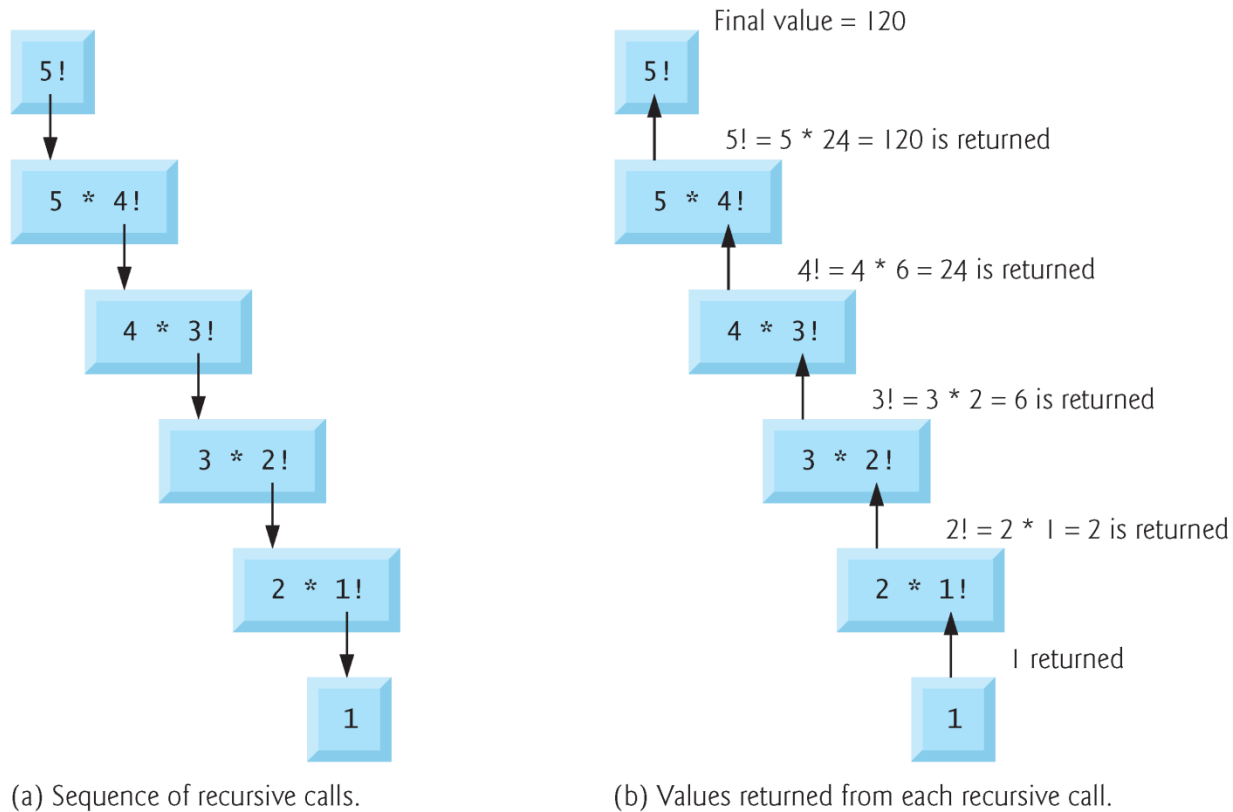


Fig. 5.13 | Recursive evaluation of 5!.



```
1  /* Fig. 5.14: fig05_14.c
2     Recursive factorial function */
3  #include <stdio.h>
4
5  long factorial( long number ); /* function prototype */
6
7  /* function main begins program execution */
8  int main( void )
9  {
10     int i; /* counter */
11
12     /* loop 11 times; during each iteration, calculate
13        factorial( i ) and display result */
14     for ( i = 0; i <= 10; i++ ) {
15         printf( "%2d! = %ld\n", i, factorial( i ) );
16     } /* end for */
17
18     return 0; /* indicates successful termination */
19 } /* end main */
20
```

Fig. 5.14 | Calculating factorials with a recursive function. (Part I of 2.)



```
21  /* recursive definition of function factorial */
22  long factorial( long number )
23  {
24      /* base case */
25      if ( number <= 1 ) {
26          return 1;
27      } /* end if */
28      else { /* recursive step */
29          return ( number * factorial( number - 1 ) );
30      } /* end else */
31  } /* 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
```

Fig. 5.14 | Calculating factorials with a recursive function. (Part 2 of 2.)



5.14 Recursion (Cont.)

- ▶ The conversion specifier `%ld` is used to print `long` values.
- ▶ Unfortunately, the `factorial` function produces large values so quickly that even `long int` does not help us print many factorial values before the size of a `long int` variable is exceeded.
- ▶ As we'll explore in the exercises, `double` may ultimately be needed by the user desiring to calculate factorials of larger numbers.



5.14 Recursion (Cont.)

- ▶ This points to a weakness in C (and most other procedural programming languages), namely that the language is not easily extended to handle the unique requirements of various applications.
- ▶ As we'll see later in the book, C++ is an extensible language that, through “classes,” allows us to create arbitrarily large integers if we wish.



Common Programming Error 5.13

Forgetting to return a value from a recursive function when one is needed.



Common Programming Error 5.14

Either omitting the base case, or writing the recursion step incorrectly so that it does not converge on the base case, will cause infinite recursion, eventually exhausting memory. This is analogous to the problem of an infinite loop in an iterative (nonrecursive) solution. Infinite recursion can also be caused by providing an unexpected input.



5.15 Example Using Recursion: Fibonacci Series

- ▶ The Fibonacci series
 - 0, 1, 1, 2, 3, 5, 8, 13, 21, ...
- ▶ begins with 0 and 1 and has the property that each subsequent Fibonacci number is the sum of the previous two Fibonacci numbers.
- ▶ The series occurs in nature and, in particular, describes a form of spiral.
- ▶ The ratio of successive Fibonacci numbers converges to a constant value of 1.618....



5.15 Example Using Recursion: Fibonacci Series (Cont.)

- ▶ This number, too, repeatedly occurs in nature and has been called the golden ratio or the golden mean.
- ▶ Humans tend to find the golden mean aesthetically pleasing.
- ▶ Architects often design windows, rooms, and buildings whose length and width are in the ratio of the golden mean.
- ▶ Postcards are often designed with a golden mean length/width ratio.



5.15 Example Using Recursion: Fibonacci Series (Cont.)

- ▶ The Fibonacci series may be defined recursively as follows:

`fibonacci(0) = 0`

`fibonacci(1) = 1`

`fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2)`

- ▶ Figure 5.15 calculates the n^{th} Fibonacci number recursively using function `fibonacci`.
- ▶ Notice that Fibonacci numbers tend to become large quickly.
- ▶ Therefore, we've chosen the data type `long` for the parameter type and the return type in function `fibonacci`.
- ▶ In Fig. 5.15, each pair of output lines shows a separate run of the program.



```
1  /* Fig. 5.15: fig05_15.c
2     Recursive fibonacci function */
3  #include <stdio.h>
4
5  long fibonacci( long n ); /* function prototype */
6
7  /* function main begins program execution */
8  int main( void )
9  {
10     long result; /* fibonacci value */
11     long number; /* number input by user */
12
13     /* obtain integer from user */
14     printf( "Enter an integer: " );
15     scanf( "%ld", &number );
16
17     /* calculate fibonacci value for number input by user */
18     result = fibonacci( number );
19
20     /* display result */
21     printf( "Fibonacci( %ld ) = %ld\n", number, result );
22     return 0; /* indicates successful termination */
23 } /* end main */
24
```

Fig. 5.15 | Recursively generating Fibonacci numbers. (Part I of 4.)



```
25  /* Recursive definition of function fibonacci */
26  long fibonacci( long n )
27  {
28      /* base case */
29      if ( n == 0 || n == 1 ) {
30          return n;
31      } /* end if */
32      else { /* recursive step */
33          return fibonacci( n - 1 ) + fibonacci( n - 2 );
34      } /* end else */
35  } /* end function fibonacci */
```

Enter an integer: 0
Fibonacci(0) = 0

Enter an integer: 1
Fibonacci(1) = 1

Enter an integer: 2
Fibonacci(2) = 1

Fig. 5.15 | Recursively generating Fibonacci numbers. (Part 2 of 4.)



```
Enter an integer: 3  
Fibonacci( 3 ) = 2
```

```
Enter an integer: 4  
Fibonacci( 4 ) = 3
```

```
Enter an integer: 5  
Fibonacci( 5 ) = 5
```

```
Enter an integer: 6  
Fibonacci( 6 ) = 8
```

```
Enter an integer: 10  
Fibonacci( 10 ) = 55
```

Fig. 5.15 | Recursively generating Fibonacci numbers. (Part 3 of 4.)



```
Enter an integer: 20  
Fibonacci( 20 ) = 6765
```

```
Enter an integer: 30  
Fibonacci( 30 ) = 832040
```

```
Enter an integer: 35  
Fibonacci( 35 ) = 9227465
```

Fig. 5.15 | Recursively generating Fibonacci numbers. (Part 4 of 4.)



5.15 Example Using Recursion: Fibonacci Series (Cont.)

- ▶ The call to `fibonacci` from `main` is not a recursive call (line 18), but all subsequent calls to `fibonacci` are recursive (line 33).
- ▶ Each time `fibonacci` is invoked, it immediately tests for the base case—`n` is equal to 0 or 1.
- ▶ If this is true, `n` is returned.
- ▶ Interestingly, if `n` is greater than 1, the recursion step generates two recursive calls, each of which is for a slightly simpler problem than the original call to `fibonacci`.
- ▶ Figure 5.16 shows how function `fibonacci` would evaluate `fibonacci(3)`.

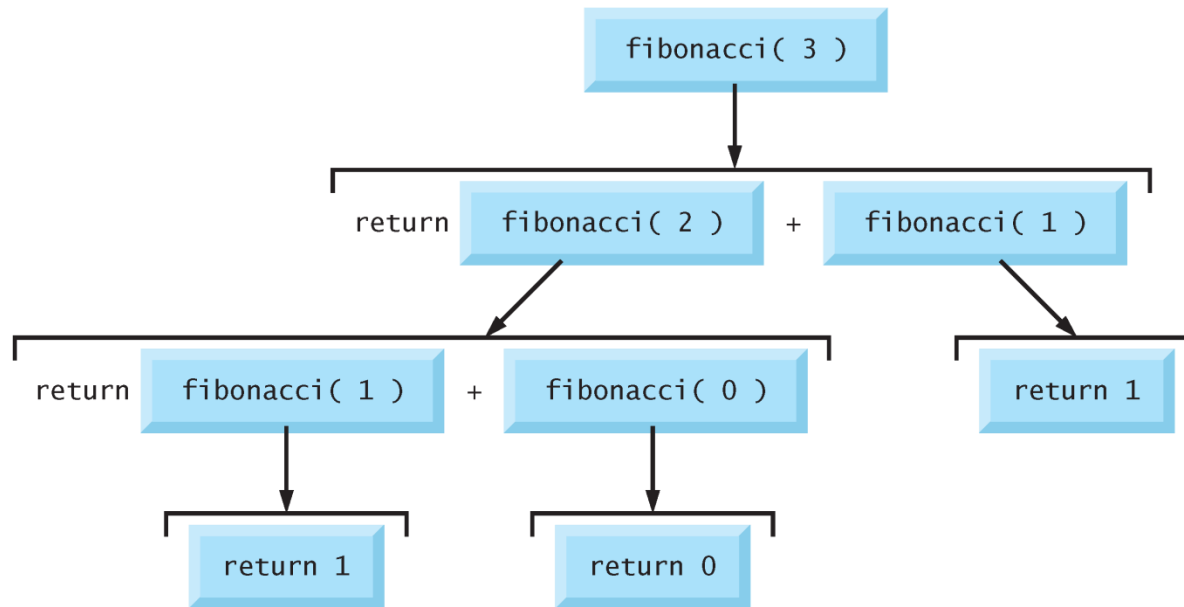


Fig. 5.16 | Set of recursive calls for `fibonacci( 3 )`.



5.15 Example Using Recursion: Fibonacci Series (Cont.)

- ▶ This figure raises some interesting issues about the order in which C compilers will evaluate the operands of operators.
- ▶ This is a different issue from the order in which operators are applied to their operands, namely the order dictated by the rules of operator precedence.
- ▶ From Fig. 5.16 it appears that while evaluating `fibonacci(3)`, two recursive calls will be made, namely `fibonacci(2)` and `fibonacci(1)`.
- ▶ But in what order will these calls be made? Most programmers simply assume the operands will be evaluated left to right.



5.15 Example Using Recursion: Fibonacci Series (Cont.)

- ▶ Strangely, Standard C does not specify the order in which the operands of most operators (including `+`) are to be evaluated.
- ▶ Therefore, you may make no assumption about the order in which these calls will execute.
- ▶ The calls could in fact execute `fibonacci(2)` first and then `fibonacci(1)`, or the calls could execute in the reverse order, `fibonacci(1)` then `fibonacci(2)`.
- ▶ In this program and in most other programs, it turns out the final result would be the same.



5.15 Example Using Recursion: Fibonacci Series (Cont.)

- ▶ But in some programs the evaluation of an operand may have side effects that could affect the final result of the expression.
- ▶ Of C's many operators, Standard C specifies the order of evaluation of the operands of only four operators—namely `&&`, `||`, the comma `(,)` operator and `? :`.
- ▶ The first three of these are binary operators whose two operands are guaranteed to be evaluated left to right.



5.15 Example Using Recursion: Fibonacci Series (Cont.)

- ▶ *[Note: The commas used to separate the arguments in a function call are not comma operators.] The last operator is C's only ternary operator.*
- ▶ Its leftmost operand is always evaluated first; if the leftmost operand evaluates to nonzero, the middle operand is evaluated next and the last operand is ignored; if the leftmost operand evaluates to zero, the third operand is evaluated next and the middle operand is ignored.



Common Programming Error 5.15

Writing programs that depend on the order of evaluation of the operands of operators other than `&&`, `||`, `?:`, and the comma `(,)` operator can lead to errors because compilers may not necessarily evaluate the operands in the order you expect.



Portability Tip 5.2

Programs that depend on the order of evaluation of the operands of operators other than `&&`, `||`, `?:`, and the comma `(,)` operator can function differently on systems with different compilers.



5.15 Example Using Recursion: Fibonacci Series (Cont.)

- ▶ A word of caution is in order about recursive programs like the one we use here to generate Fibonacci numbers.
- ▶ Each level of recursion in the `fibonacci` function has a doubling effect on the number of calls; i.e., the number of recursive calls that will be executed to calculate the n^{th} Fibonacci number is on the order of 2^n .
- ▶ This rapidly gets out of hand.
- ▶ Calculating only the 20th Fibonacci number would require on the order of 2^{20} or about a million calls, calculating the 30th Fibonacci number would require on the order of 2^{30} or about a billion calls, and so on.



5.15 Example Using Recursion: Fibonacci Series (Cont.)

- ▶ Computer scientists refer to this as exponential complexity.
- ▶ Problems of this nature humble even the world's most powerful computers!
- ▶ Complexity issues in general, and exponential complexity in particular, are discussed in detail in the upper-level computer science curriculum course generally called “Algorithms.”



Performance Tip 5.4

Avoid Fibonacci-style recursive programs which result in an exponential “explosion” of calls.



5.15 Example Using Recursion: Fibonacci Series (Cont.)

- ▶ The example we showed in this section used an intuitively appealing solution to calculate Fibonacci numbers, but there are better approaches.
- ▶ Exercise 6.48 asks you to investigate recursion in more depth and propose alternate approaches to implementing the recursive Fibonacci algorithm.



5.16 Recursion vs. Iteration

- ▶ In the previous sections, we studied two functions that can easily be implemented either recursively or iteratively.
- ▶ In this section, we compare the two approaches and discuss why you might choose one approach over the other in a particular situation.
- ▶ Both iteration and recursion are based on a control structure: Iteration uses a repetition structure; recursion uses a selection structure.
- ▶ Both iteration and recursion involve repetition: Iteration explicitly uses a repetition structure; recursion achieves repetition through repeated function calls.



5.16 Recursion vs. Iteration (Cont.)

- ▶ Iteration and recursion each involve a termination test: Iteration terminates when the loop-continuation condition fails; recursion terminates when a base case is recognized.
- ▶ Iteration with counter-controlled repetition and recursion each gradually approach termination: Iteration keeps modifying a counter until the counter assumes a value that makes the loop-continuation condition fail; recursion keeps producing simpler versions of the original problem until the base case is reached.



5.16 Recursion vs. Iteration (Cont.)

- ▶ Both iteration and recursion can occur infinitely: An infinite loop occurs with iteration if the loop-continuation test never becomes false; infinite recursion occurs if the recursion step does not reduce the problem each time in a manner that converges on the base case.
- ▶ Recursion has many negatives.
- ▶ It repeatedly invokes the mechanism, and consequently the overhead, of function calls.
- ▶ This can be expensive in both processor time and memory space.



5.16 Recursion vs. Iteration (Cont.)

- ▶ Each recursive call causes another copy of the function (actually only the function's variables) to be created; this can consume considerable memory.
- ▶ Iteration normally occurs within a function, so the overhead of repeated function calls and extra memory assignment is omitted.
- ▶ So why choose recursion?



Software Engineering Observation 5.15

Any problem that can be solved recursively can also be solved iteratively (nonrecursively). A recursive approach is normally chosen in preference to an iterative approach when the recursive approach more naturally mirrors the problem and results in a program that is easier to understand and debug. Another reason to choose a recursive solution is that an iterative solution may not be apparent.



Performance Tip 5.5

Avoid using recursion in performance situations. Recursive calls take time and consume additional memory.



Common Programming Error 5.16

Accidentally having a nonrecursive function call itself either directly, or indirectly through another function.



5.16 Recursion vs. Iteration (Cont.)

- ▶ Figure 5.17 summarizes by chapter the 31 recursion examples and exercises in the text.



Chapter	Recursion examples and exercises
<i>Chapter 5</i>	Factorial function Fibonacci function Greatest common divisor Sum of two integers Multiply two integers Raising an integer to an integer power Towers of Hanoi Recursive main Printing keyboard inputs in reverse Visualizing recursion
<i>Chapter 6</i>	Sum the elements of an array Print an array Print an array backward Print a string backward Check if a string is a palindrome Minimum value in an array Linear search Binary search

Fig. 5.17 | Recursion examples and exercises in the text. (Part I of 2.)



Chapter	Recursion examples and exercises
<i>Chapter 7</i>	Eight Queens Maze traversal
<i>Chapter 8</i>	Printing a string input at the keyboard backward
<i>Chapter 12</i>	Linked list insert Linked list delete Search a linked list Print a linked list backward Binary tree insert Preorder traversal of a binary tree Inorder traversal of a binary tree Postorder traversal of a binary tree
<i>Appendix F</i>	Selection sort Quicksort

Fig. 5.17 | Recursion examples and exercises in the text. (Part 2 of 2.)



5.16 Recursion vs. Iteration (Cont.)

- ▶ Let's close this chapter with some observations that we make repeatedly throughout the book.
- ▶ Good software engineering is important.
- ▶ High performance is important.
- ▶ Unfortunately, these goals are often at odds with one another.
- ▶ Good software engineering is key to making more manageable the task of developing the larger and more complex software systems we need.
- ▶ High performance is key to realizing the systems of the future that will place ever greater computing demands on hardware.
- ▶ Where do functions fit in here?



Performance Tip 5.6

Functionalizing programs in a neat, hierarchical manner promotes good software engineering. But it has a price. A heavily functionalized program—as compared to a monolithic (i.e., one-piece) program without functions—makes potentially large numbers of function calls, and these consume execution time on a computer’s processor(s). So, although monolithic programs may perform better, they’re more difficult to program, test, debug, maintain, and evolve.