



C Pointers

C How to Program, 6/e



OBJECTIVES

In this chapter, you'll learn:

- Pointers and pointer operators.
- To use pointers to pass arguments to functions by reference.
- The close relationships among pointers, arrays and strings.
- To use pointers to functions.
- To define and use arrays of strings.



- 7.1 Introduction
- 7.2 Pointer Variable Definitions and Initialization
- 7.3 Pointer Operators
- 7.4 Passing Arguments to Functions by Reference
- 7.5 Using the `const` Qualifier with Pointers
- 7.6 Bubble Sort Using Call-by-Reference
- 7.7 `sizeof` Operator
- 7.8 Pointer Expressions and Pointer Arithmetic
- 7.9 Relationship between Pointers and Arrays
- 7.10 Arrays of Pointers
- 7.11 Case Study: Card Shuffling and Dealing Simulation
- 7.12 Pointers to Functions



7.1 Introduction

- ▶ In this chapter, we discuss one of the most powerful features of the C programming language, the **pointer**.
- ▶ Pointers enable programs to simulate call-by-reference and to create and manipulate dynamic data structures, i.e., data structures that can grow and shrink at execution time, such as linked lists, queues, stacks and trees.
- ▶ Chapter 10 examines the use of pointers with structures.
- ▶ Chapter 12 introduces dynamic memory management techniques and presents examples of creating and using dynamic data structures.



7.2 Pointer Variable Definitions and Initialization

- ▶ Pointers are variables whose values are memory addresses.
- ▶ Normally, a variable directly contains a specific value.
- ▶ A pointer, on the other hand, contains an address of a variable that contains a specific value.
- ▶ In this sense, a variable name directly references a value, and a pointer indirectly references a value (Fig. 7.1).
- ▶ Referencing a value through a pointer is called **indirection**.

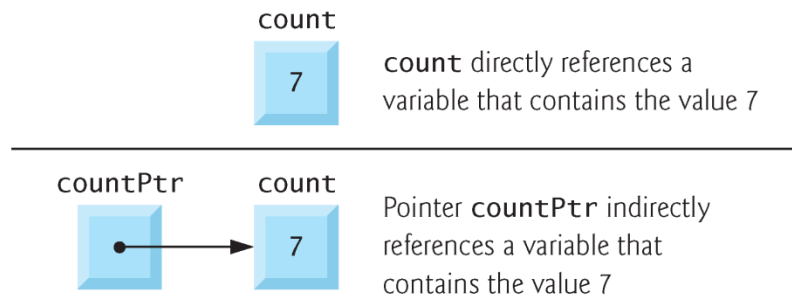


Fig. 7.1 | Directly and indirectly referencing a variable.



7.2 Pointer Variable Definitions and Initialization (cont.)

- ▶ Pointers, like all variables, must be defined before they can be used.
- ▶ The definition
 - `int *countPtr, count;`
specifies that variable `countPtr` is of type `int *` (i.e., a pointer to an integer) and is read, “`countPtr` is a pointer to `int`” or “`countPtr` points to an object of type `int`.” Also, the variable `count` is defined to be an `int`, not a pointer to an `int`.
- ▶ The `*` only applies to `countPtr` in the definition.
- ▶ When `*` is used in this manner in a definition, it indicates that the variable being defined is a pointer.
- ▶ Pointers can be defined to point to objects of any type.



Common Programming Error 7.1

The asterisk () notation used to declare pointer variables does not distribute to all variable names in a declaration. Each pointer must be declared with the * prefixed to the name; e.g., if you wish to declare xPtr and yPtr as int pointers, use `int *xPtr, *yPtr;`*



Common Programming Error 7.2

Include the letters `ptr` in pointer variable names to make it clear that these variables are pointers and thus need to be handled appropriately.



7.2 Pointer Variable Definitions and Initialization (cont.)

- ▶ Pointers should be initialized either when they're defined or in an assignment statement.
- ▶ A pointer may be initialized to `NULL`, `0` or an address.
- ▶ A pointer with the value `NULL` points to nothing.
- ▶ `NULL` is a symbolic constant defined in the `<stddef.h>` header (and several other headers, such as `<stdio.h>`).
- ▶ Initializing a pointer to `0` is equivalent to initializing a pointer to `NULL`, but `NULL` is preferred.
- ▶ When `0` is assigned, it's first converted to a pointer of the appropriate type.
- ▶ The value `0` is the only integer value that can be assigned directly to a pointer variable.



Error-Prevention Tip 7.1

Initialize pointers to prevent unexpected results.

7.3 Pointer Operators

- ▶ The `&`, or **address operator**, is a unary operator that returns the address of its operand.
- ▶ For example, assuming the definitions
 - `int y = 5;`
`int *yPtr;`the statement
 - `yPtr = &y;`assigns the address of the variable `y` to pointer variable `yPtr`.
- ▶ Variable `yPtr` is then said to “point to” `y`.
- ▶ Figure 7.2 shows a schematic representation of memory after the preceding assignment is executed.

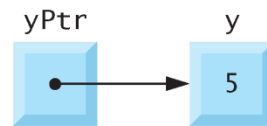


Fig. 7.2 | Graphical representation of a pointer pointing to an integer variable in memory.



7.3 Pointer Operators (Cont.)

- ▶ Figure 7.3 shows the representation of the pointer in memory, assuming that integer variable `y` is stored at location 600000, and pointer variable `yPtr` is stored at location 500000.
- ▶ The operand of the address operator must be a variable; the address operator cannot be applied to constants, to expressions or to variables declared with the storage-class `register`.



Fig. 7.3 | Representation of `y` and `yPtr` in memory.



7.3 Pointer Operators (Cont.)

- ▶ The unary `*` operator, commonly referred to as the **indirection operator** or **dereferencing operator**, returns the value of the object to which its operand (i.e., a pointer) points.
- ▶ For example, the statement
 - `printf( "%d", *yPtr );`
prints the value of variable `y`, namely 5.
- ▶ Using `*` in this manner is called **dereferencing a pointer**.



Common Programming Error 7.3

Dereferencing a pointer that has not been properly initialized or that has not been assigned to point to a specific location in memory is an error. This could cause a fatal execution-time error, or it could accidentally modify important data and allow the program to run to completion with incorrect results.



7.3 Pointer Operators (Cont.)

- ▶ Figure 7.4 demonstrates the pointer operators `&` and `*`.
- ▶ The `printf` conversion specifier `%p` outputs the memory location as a hexadecimal integer on most platforms.
- ▶ (See Appendix C, Number Systems, for more information on hexadecimal integers.) Notice that the address of `a` and the value of `aPtr` are identical in the output, thus confirming that the address of `a` is indeed assigned to the pointer variable `aPtr` (line 11).
- ▶ The `&` and `*` operators are complements of one another—when they're both applied consecutively to `aPtr` in either order (line 21), the same result is printed.
- ▶ Figure 7.5 lists the precedence and associativity of the operators introduced to this point.



```
1  /* Fig. 7.4: fig07_04.c
2     Using the & and * operators */
3  #include <stdio.h>
4
5  int main( void )
6  {
7     int a; /* a is an integer */
8     int *aPtr; /* aPtr is a pointer to an integer */
9
10    a = 7;
11    aPtr = &a; /* aPtr set to address of a */
12
13    printf( "The address of a is %p"
14           "\nThe value of aPtr is %p", &a, aPtr );
15
16    printf( "\n\nThe value of a is %d"
17           "\nThe value of *aPtr is %d", a, *aPtr );
18
19    printf( "\n\nShowing that * and & are complements of "
20           "each other\n&*aPtr = %p"
21           "\n*&aPtr = %p\n", &*aPtr, *&aPtr );
22    return 0; /* indicates successful termination */
23 } /* end main */
```

Fig. 7.4 | Using the & and * pointer operators. (Part I of 2.)



```
The address of a is 0012FF7C  
The value of aPtr is 0012FF7C
```

```
The value of a is 7  
The value of *aPtr is 7
```

```
Showing that * and & are complements of each other.  
&*aPtr = 0012FF7C  
*&aPtr = 0012FF7C
```

Fig. 7.4 | Using the & and * pointer operators. (Part 2 of 2.)



Operators	Associativity	Type
() []	left to right	highest
+ - ++ -- ! * & (type)	right to left	unary
* / %	left to right	multiplicative
+ -	left to right	additive
< <= > >=	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

Fig. 7.5 | Operator precedence and associativity.

7.4 Passing Arguments to Functions by Reference



- ▶ There are two ways to pass arguments to a function—**call-by-value** and **call-by-reference**.
- ▶ All arguments in C are passed by value.
- ▶ Many functions require the capability to modify one or more variables in the caller or to pass a pointer to a large data object to avoid the overhead of passing the object by value (which incurs the overhead of making a copy of the object).
- ▶ For these purposes, C provides the capabilities for **simulating call-by-reference**.
- ▶ In C, you use pointers and the indirection operator to simulate call-by-reference.

7.4 Passing Arguments to Functions by Reference (Cont.)



- ▶ When calling a function with arguments that should be modified, the addresses of the arguments are passed.
- ▶ This is normally accomplished by applying the address operator (&) to the variable (in the caller) whose value will be modified.
- ▶ As we saw in Chapter 6, arrays are not passed using operator & because C automatically passes the starting location in memory of the array (the name of an array is equivalent to `&arrayName[0]`).
- ▶ When the address of a variable is passed to a function, the indirection operator (*) may be used in the function to modify the value at that location in the caller's memory.

7.4 Passing Arguments to Functions by Reference (Cont.)



- ▶ The programs in Fig. 7.6 and Fig. 7.7 present two versions of a function that cubes an integer—`cubeByValue` and `cubeByReference`.
- ▶ Figure 7.6 passes the variable `number` to function `cubeByValue` using call-by-value (line 14).
- ▶ The `cubeByValue` function cubes its argument and passes the new value back to `main` using a `return` statement.
- ▶ The new value is assigned to `number` in `main` (line 14).



```
1  /* Fig. 7.6: fig07_06.c
2     Cube a variable using call-by-value */
3  #include <stdio.h>
4
5  int cubeByValue( int n ); /* prototype */
6
7  int main( void )
8  {
9     int number = 5; /* initialize number */
10
11     printf( "The original value of number is %d", number );
12
13     /* pass number by value to cubeByValue */
14     number = cubeByValue( number );
15
16     printf( "\nThe new value of number is %d\n", number );
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 */
```

Fig. 7.6 | Cube a variable using call-by-value. (Part 1 of 2.)



The original value of number is 5
The new value of number is 125

Fig. 7.6 | Cube a variable using call-by-value. (Part 2 of 2.)

7.4 Passing Arguments to Functions by Reference (Cont.)



- ▶ Figure 7.7 passes the variable `number` using call-by-reference (line 15)—the address of `number` is passed—to function `cubeByReference`.
- ▶ Function `cubeByReference` takes as a parameter a pointer to an `int` called `nPtr` (line 22).
- ▶ The function dereferences the pointer and cubes the value to which `nPtr` points (line 24), then assigns the result to `*nPtr` (which is really `number` in `main`), thus changing the value of `number` in `main`.
- ▶ Figure 7.8 and Fig. 7.9 analyze graphically the programs in Fig. 7.6 and Fig. 7.7, respectively.



```
1  /* Fig. 7.7: fig07_07.c
2     Cube a variable using call-by-reference with a pointer argument */
3
4  #include <stdio.h>
5
6  void cubeByReference( int *nPtr ); /* prototype */
7
8  int main( void )
9  {
10     int number = 5; /* initialize number */
11
12     printf( "The original value of number is %d", number );
13
14     /* pass address of number to cubeByReference */
15     cubeByReference( &number );
16
17     printf( "\nThe new value of number is %d\n", number );
18     return 0; /* indicates successful termination */
19 } /* end main */
20
```

Fig. 7.7 | Cube a variable using call-by-reference with a pointer argument. (Part I of 2.)



```
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

Fig. 7.7 | Cube a variable using call-by-reference with a pointer argument. (Part 2 of 2.)

7.4 Passing Arguments to Functions by Reference (Cont.)



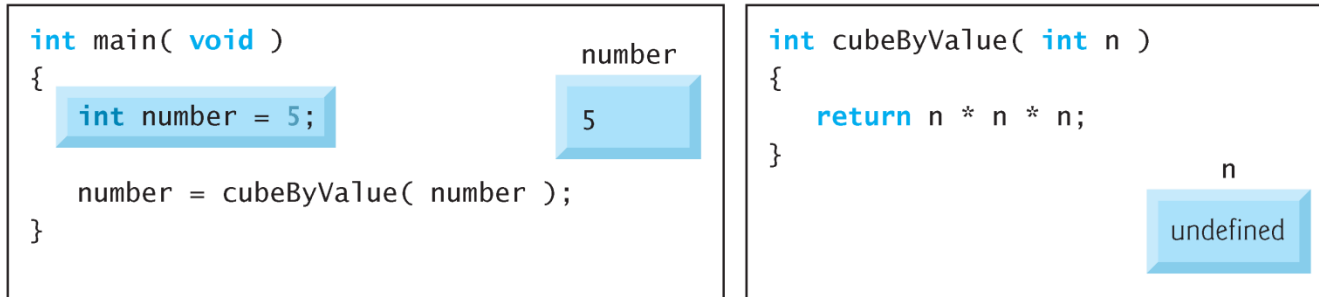
- ▶ A function receiving an address as an argument must define a pointer parameter to receive the address.
- ▶ For example, in Fig. 7.7 the header for function **cubeByReference** (line 22) is:
 - **void cubeByReference(int *nPtr)**
- ▶ The header specifies that **cubeByReference** receives the address of an integer variable as an argument, stores the address locally in **nPtr** and does not return a value.
- ▶ The function prototype for **cubeByReference** contains **int *** in parentheses.
- ▶ Names included for documentation purposes are ignored by the C compiler.

7.4 Passing Arguments to Functions by Reference (Cont.)



- ▶ In the function header and in the prototype for a function that expects a single-subscripted array as an argument, the pointer notation in the parameter list of function `cubeByReference` may be used.
- ▶ The compiler does not differentiate between a function that receives a pointer and a function that receives a single-subscripted array.
- ▶ This, of course, means that the function must “know” when it’s receiving an array or simply a single variable for which it is to perform call by reference.
- ▶ When the compiler encounters a function parameter for a single-subscripted array of the form `int b[]`, the compiler converts the parameter to the pointer notation `int *b`.
- ▶ The two forms are interchangeable.

Step 1: Before `main` calls `cubeByValue`:



Step 2: After `cubeByValue` receives the call:

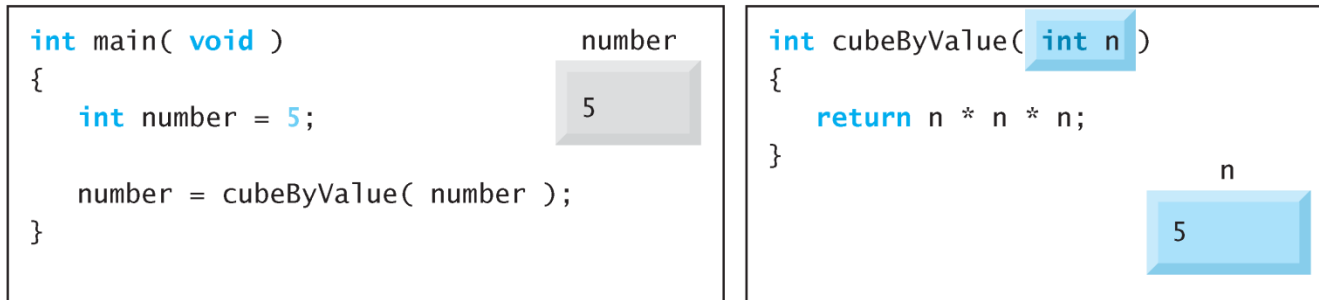
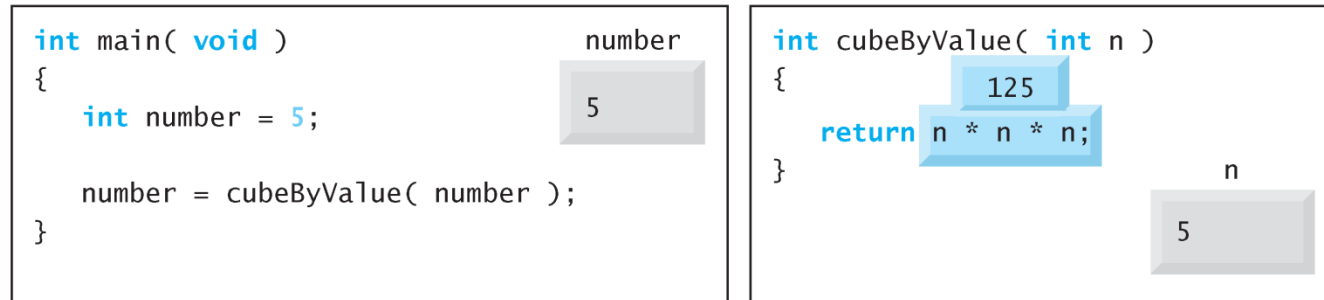


Fig. 7.8 | Analysis of a typical call-by-value. (Part I of 3.)

Step 3: After `cubeByValue` cubes parameter `n` and before `cubeByValue` returns to `main`:



Step 4: After `cubeByValue` returns to `main` and before assigning the result to `number`:

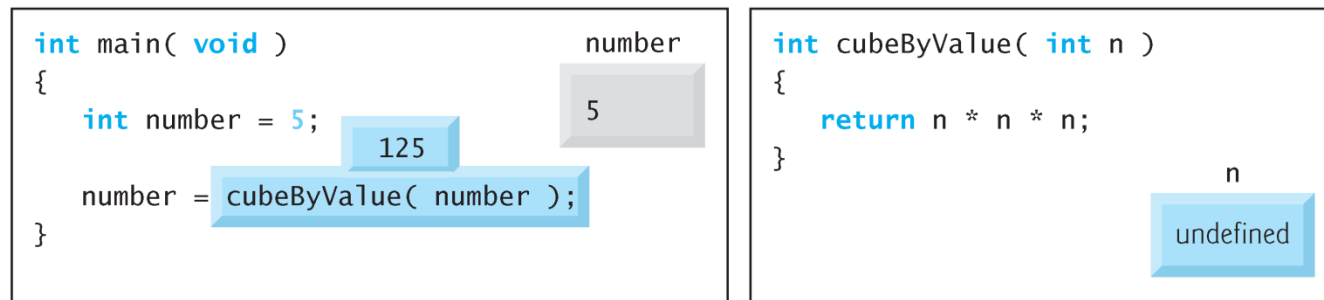


Fig. 7.8 | Analysis of a typical call-by-value. (Part 2 of 3.)

Step 5: After `main` completes the assignment to `number`:

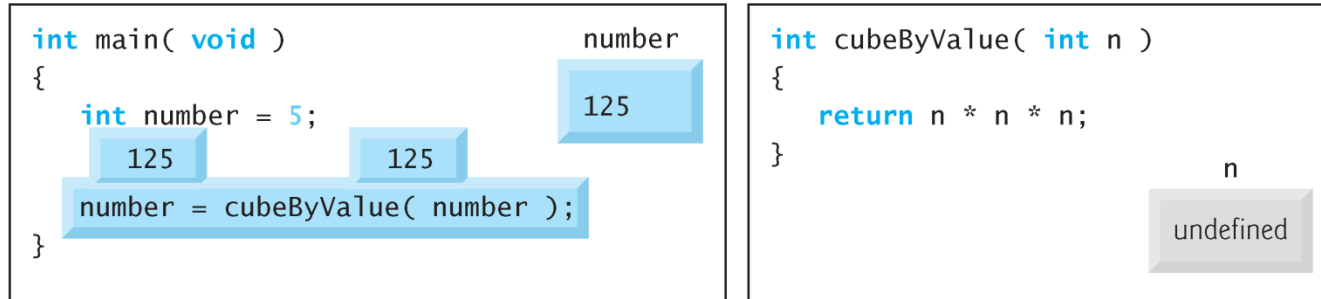
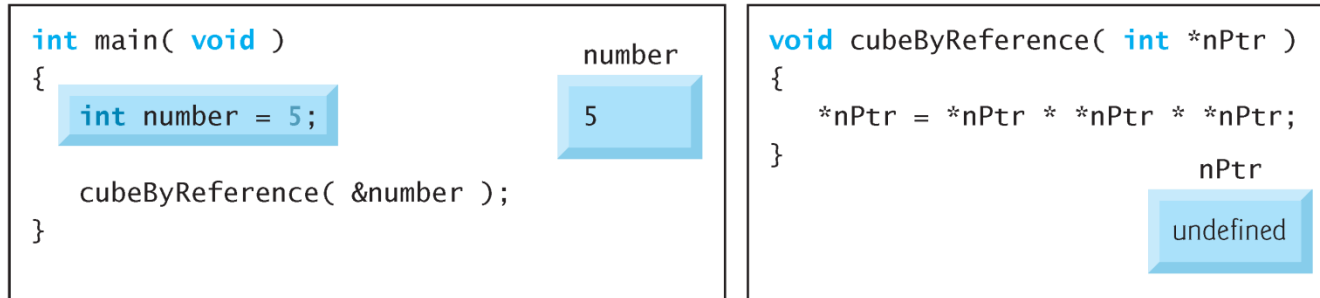


Fig. 7.8 | Analysis of a typical call-by-value. (Part 3 of 3.)

Step 1: Before `main` calls `cubeByReference`:



Step 2: After `cubeByReference` receives the call and before `*nPtr` is cubed:

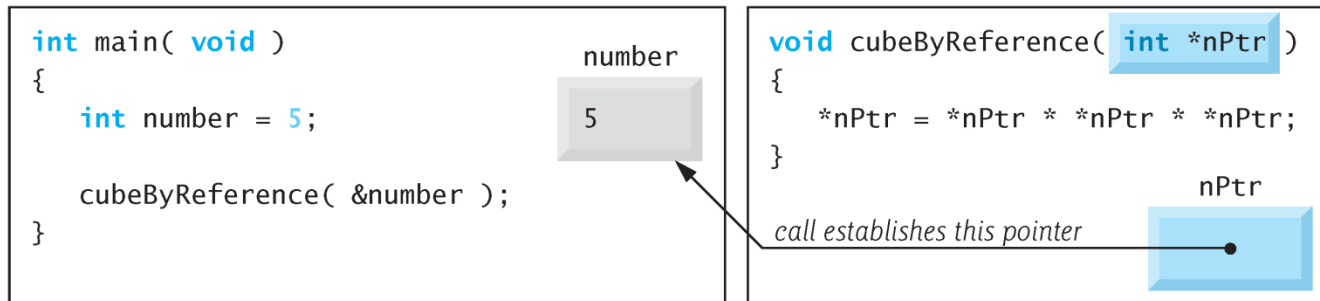


Fig. 7.9 | Analysis of a typical call-by-reference with a pointer argument.

Step 3: After `*nPtr` is cubed and before program control returns to `main`:

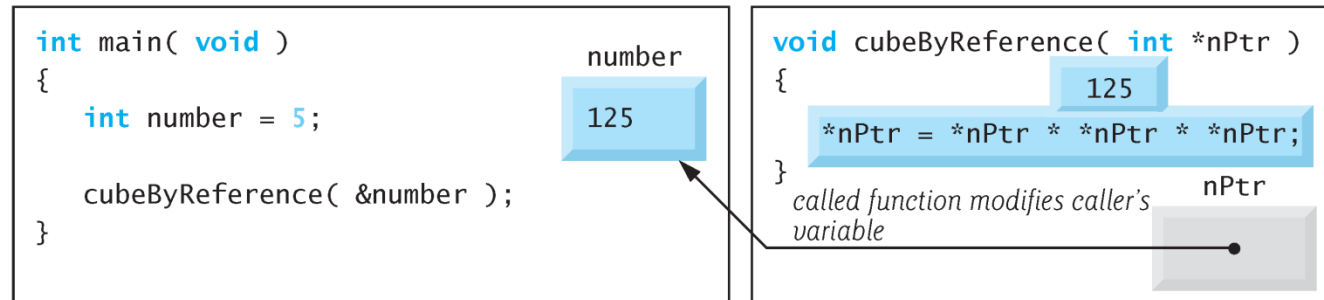


Fig. 7.9 | Analysis of a typical call-by-reference with a pointer argument.



Error-Prevention Tip 7.2

Use call-by-value to pass arguments to a function unless the caller explicitly requires the called function to modify the value of the argument variable in the caller's environment. This prevents accidental modification of the caller's arguments and is another example of the principle of least privilege.



7.5 Using the `const` Qualifier with Pointers

- ▶ The `const` qualifier enables you to inform the compiler that the value of a particular variable should not be modified.



Software Engineering Observation 7.1

The `const` qualifier can be used to enforce the principle of least privilege. Using the principle of least privilege to properly design software reduces debugging time and improper side effects, making a program easier to modify and maintain.



Portability Tip 7.1

Although `const` is well defined in Standard C, some compilers do not enforce it.



7.5 Using the `const` Qualifier with Pointers (`Const.`)

- ▶ Over the years, a large base of legacy code was written in early versions of C that did not use `const` because it was not available.
- ▶ For this reason, there are significant opportunities for improvement in the reengineering of old C code.
- ▶ Six possibilities exist for using (or not using) `const` with function parameters—two with call-by-value parameter passing and four with call-by-reference parameter passing.
- ▶ How do you choose one of the six possibilities? Let the [principle of least privilege](#) be your guide.
- ▶ Always award a function enough access to the data in its parameters to accomplish its specified task, but no more.



7.5 Using the const Qualifier with Pointers (Const.)

- ▶ In Chapter 5, we explained that all calls in C are call-by-value—a copy of the argument in the function call is made and passed to the function.
- ▶ If the copy is modified in the function, the original value in the caller does not change.
- ▶ In many cases, a value passed to a function is modified so the function can accomplish its task.
- ▶ However, in some instances, the value should not be altered in the called function, even though it manipulates only a copy of the original value.
- ▶ Consider a function that takes a single-subscripted array and its size as arguments and prints the array.



7.5 Using the const Qualifier with Pointers (Const.)

- ▶ Such a function should loop through the array and output each array element individually.
- ▶ The size of the array is used in the function body to determine the high subscript of the array, so the loop can terminate when the printing is completed.
- ▶ Neither the size of the array nor its contents should change in the function body.



Error-Prevention Tip 7.3

If a variable does not (or should not) change in the body of a function to which it's passed, the variable should be declared `const` to ensure that it's not accidentally modified.



7.5 Using the `const` Qualifier with Pointers (`Const.`)

- ▶ If an attempt is made to modify a value that is declared `const`, the compiler catches it and issues either a warning or an error, depending on the particular compiler.



Software Engineering Observation 7.2

Only one value in a calling function can be altered when using call-by-value. That value must be assigned from the return value of the function to a variable in the caller. To modify multiple variables from a calling function in a called function, use call-by-reference.



Error-Prevention Tip 7.4

Before using a function, check its function prototype to determine if the function is able to modify the values passed to it.



Common Programming Error 7.4

Being unaware that a function is expecting pointers as arguments for call-by-reference and passing arguments call-by-value. Some compilers take the values assuming they're pointers and dereference the values as pointers. At runtime, memory-access violations or segmentation faults are often generated. Other compilers catch the mismatch in types between arguments and parameters and generate error messages.



7.5 Using the const Qualifier with Pointers (Const.)

- ▶ There are four ways to pass a pointer to a function: a non-constant pointer to non-constant data, a constant pointer to nonconstant data, a non-constant pointer to constant data, and a constant pointer to constant data.
- ▶ Each of the four combinations provides different access privileges.
- ▶ These are discussed in the next several examples.



7.5 Using the `const` Qualifier with Pointers (`Const.`)

- ▶ The highest level of data access is granted by a non-constant pointer to non-constant data.
- ▶ In this case, the data can be modified through the dereferenced pointer, and the pointer can be modified to point to other data items.
- ▶ A declaration for a non-constant pointer to non-constant data does not include **`const`**.
- ▶ Such a pointer might be used to receive a string as an argument to a function that uses **pointer arithmetic** to process (and possibly modify) each character in the string.



7.5 Using the `const` Qualifier with Pointers (`Const.`)

- ▶ Function `convertToUppercase` of Fig. 7.10 declares its parameter, a non-constant pointer to non-constant data called `SPtr` (`char *SPtr`), in line 21.
- ▶ The function processes the array `string` (pointed to by `SPtr`) one character at a time using pointer arithmetic.
- ▶ C standard library function `islower` (called in line 25) tests the character contents of the address pointed to by `SPtr`.
- ▶ If a character is in the range `a` to `z`, `islower` returns true and C standard library function `toupper` (line 26) is called to convert the character to its corresponding uppercase letter; otherwise, `islower` returns false and the next character in the string is processed.
- ▶ Line 29 moves the pointer to the next character in the string.



```
1  /* Fig. 7.10: fig07_10.c
2     Converting a string to uppercase using a
3     non-constant pointer to non-constant data */
4
5  #include <stdio.h>
6  #include <ctype.h>
7
8  void convertToUppercase( char *sPtr ); /* prototype */
9
10 int main( void )
11 {
12     char string[] = "characters and $32.98"; /* initialize char array */
13
14     printf( "The string before conversion is: %s", string );
15     convertToUppercase( string );
16     printf( "\nThe string after conversion is: %s\n", string );
17     return 0; /* indicates successful termination */
18 } /* end main */
19
```

Fig. 7.10 | Converting a string to uppercase using a non-constant pointer to non-constant data. (Part 1 of 2.)



```
20  /* convert string to uppercase letters */
21  void convertToUpper( char *sPtr )
22  {
23      while ( *sPtr != '\0' ) { /* current character is not '\0' */
24
25          if ( islower( *sPtr ) ) { /* if character is lowercase, */
26              *sPtr = toupper( *sPtr ); /* convert to uppercase */
27          } /* end if */
28
29          ++sPtr; /* move sPtr to the next character */
30      } /* end while */
31  } /* end function convertToUpper */
```

The string before conversion is: characters and \$32.98
The string after conversion is: CHARACTERS AND \$32.98

Fig. 7.10 | Converting a string to uppercase using a non-constant pointer to non-constant data. (Part 2 of 2.)



7.5 Using the const Qualifier with Pointers (Const.)

- ▶ A non-constant pointer to constant data can be modified to point to any data item of the appropriate type, but the data to which it points cannot be modified.
- ▶ Such a pointer might be used to receive an array argument to a function that will process each element without modifying the data.



7.5 Using the `const` Qualifier with Pointers (`Const.`)

- ▶ For example, function `printCharacters` (Fig. 7.11) declares parameter `sPtr` to be of type `const char *` (line 22).
- ▶ The declaration is read from right to left as “`sPtr` is a pointer to a character constant.” The function uses a `for` statement to output each character in the string until the null character is encountered.
- ▶ After each character is printed, pointer `sPtr` is incremented to point to the next character in the string.



```
1  /* Fig. 7.11: fig07_11.c
2     Printing a string one character at a time using
3     a non-constant pointer to constant data */
4
5  #include <stdio.h>
6
7  void printCharacters( const char *sPtr );
8
9  int main( void )
10 {
11     /* initialize char array */
12     char string[] = "print characters of a string";
13
14     printf( "The string is:\n" );
15     printCharacters( string );
16     printf( "\n" );
17     return 0; /* indicates successful termination */
18 } /* end main */
19
```

Fig. 7.11 | Printing a string one character at a time using a non-constant pointer to constant data. (Part 1 of 2.)



```
20  /* sPtr cannot modify the character to which it points,  
21     i.e., sPtr is a "read-only" pointer */  
22  void printCharacters( const char *sPtr )  
23  {  
24     /* loop through entire string */  
25     for ( ; *sPtr != '\0'; sPtr++ ) { /* no initialization */  
26         printf( "%c", *sPtr );  
27     } /* end for */  
28 } /* end function printCharacters */
```

The string is:
print characters of a string

Fig. 7.11 | Printing a string one character at a time using a non-constant pointer to constant data. (Part 2 of 2.)



7.5 Using the `const` Qualifier with Pointers (`Const.`)

- ▶ Figure 7.12 illustrates the attempt to compile a function that receives a non-constant pointer (`xPtr`) to constant data.
- ▶ This function attempts to modify the data pointed to by `xPtr` in line 20—which results in a compilation error.



```
1  /* Fig. 7.12: fig07_12.c
2     Attempting to modify data through a
3     non-constant pointer to constant data. */
4  #include <stdio.h>
5  void f( const int *xPtr ); /* prototype */
6
7
8  int main( void )
9  {
10     int y; /* define y */
11
12     f( &y ); /* f attempts illegal modification */
13     return 0; /* indicates successful termination */
14 } /* end main */
15
16 /* xPtr cannot be used to modify the
17     value of the variable to which it points */
18 void f( const int *xPtr )
19 {
20     *xPtr = 100; /* error: cannot modify a const object */
21 } /* end function f */
```

Fig. 7.12 | Attempting to modify data through a non-constant pointer to constant data. (Part I of 2.)



```
Compiling...
FIG07_12.c
c:\examples\ch07\fig07_12.c(22) : error C2166: l-value specifies const object
Error executing cl.exe.

FIG07_12.exe - 1 error(s), 0 warning(s)
```

Fig. 7.12 | Attempting to modify data through a non-constant pointer to constant data. (Part 2 of 2.)



7.5 Using the const Qualifier with Pointers (Const.)

- ▶ As we know, arrays are aggregate data types that store related data items of the same type under one name.
- ▶ In Chapter 10, we'll discuss another form of aggregate data type called a **structure** (sometimes called a **record** in other languages).
- ▶ A structure is capable of storing related data items of different data types under one name (e.g., storing information about each employee of a company).
- ▶ When a function is called with an array as an argument, the array is automatically passed to the function by reference.
- ▶ However, structures are always passed by value—a copy of the entire structure is passed.



7.5 Using the const Qualifier with Pointers (Const.)

- ▶ This requires the execution-time overhead of making a copy of each data item in the structure and storing it on the computer's function call stack.
- ▶ When structure data must be passed to a function, we can use pointers to constant data to get the performance of call-by-reference and the protection of call-by-value.
- ▶ When a pointer to a structure is passed, only a copy of the address at which the structure is stored must be made.
- ▶ On a machine with 4-byte addresses, a copy of 4 bytes of memory is made rather than a copy of possibly hundreds or thousands of bytes of the structure.



Performance Tip 7.1

Pass large objects such as structures using pointers to constant data to obtain the performance benefits of call-by-reference and the security of call-by-value.



7.5 Using the `const` Qualifier with Pointers (`Const.`)

- ▶ Using pointers to constant data in this manner is an example of a **time/space trade-off**.
- ▶ If memory is low and execution efficiency is a concern, use pointers.
- ▶ If memory is in abundance and efficiency is not a major concern, pass data by value to enforce the principle of least privilege.
- ▶ Remember that some systems do not enforce `const` well, so call-by-value is still the best way to prevent data from being modified.



7.5 Using the const Qualifier with Pointers (Const.)

- ▶ A constant pointer to non-constant data always points to the same memory location, and the data at that location can be modified through the pointer.
- ▶ This is the default for an array name.
- ▶ An array name is a constant pointer to the beginning of the array.
- ▶ All data in the array can be accessed and changed by using the array name and array subscripting.
- ▶ A constant pointer to non-constant data can be used to receive an array as an argument to a function that accesses array elements using only array subscript notation.



7.5 Using the `const` Qualifier with Pointers (`Const.`)

- ▶ Pointers that are declared `const` must be initialized when they're defined (if the pointer is a function parameter, it's initialized with a pointer that is passed to the function).
- ▶ Figure 7.13 attempts to modify a constant pointer.
- ▶ Pointer `ptr` is defined in line 12 to be of type `int * const`.
- ▶ The definition is read from right to left as “`ptr` is a constant pointer to an integer.” The pointer is initialized (line 12) with the address of integer variable `x`.
- ▶ The program attempts to assign the address of `y` to `ptr` (line 15), but the compiler generates an error message.



```
1  /* Fig. 7.13: fig07_13.c
2     Attempting to modify a constant pointer to non-constant data */
3  #include <stdio.h>
4
5  int main( void )
6  {
7      int x; /* define x */
8      int y; /* define y */
9
10     /* ptr is a constant pointer to an integer that can be modified
11        through ptr, but ptr always points to the same memory location */
12     int * const ptr = &x;
13
14     *ptr = 7; /* allowed: *ptr is not const */
15     ptr = &y; /* error: ptr is const; cannot assign new address */
16     return 0; /* indicates successful termination */
17 }
```

Fig. 7.13 | Attempting to modify a constant pointer to non-constant data.



```
Compiling...
FIG07_13.c
c:\examples\ch07\FIG07_13.c(15) : error C2166: l-value specifies const object
Error executing cl.exe.

FIG07_13.exe - 1 error(s), 0 warning(s)
```

Fig. 7.13 | Attempting to modify a constant pointer to non-constant data.



7.5 Using the const Qualifier with Pointers (Const.)

- ▶ The least access privilege is granted by a constant pointer to constant data.
- ▶ Such a pointer always points to the same memory location, and the data at that memory location cannot be modified.
- ▶ This is how an array should be passed to a function that only looks at the array using array subscript notation and does not modify the array.



7.5 Using the const Qualifier with Pointers (Const.)

- ▶ Figure 7.14 defines pointer variable `ptr` (line 13) to be of type `const int *const`, which is read from right to left as “`ptr` is a constant pointer to an integer constant.”
- ▶ The figure shows the error messages generated when an attempt is made to modify the data to which `ptr` points (line 16) and when an attempt is made to modify the address stored in the pointer variable (line 17).



```
1  /* Fig. 7.14: fig07_14.c
2     Attempting to modify a constant pointer to constant data. */
3  #include <stdio.h>
4
5  int main( void )
6  {
7      int x = 5; /* initialize x */
8      int y; /* define y */
9
10     /* ptr is a constant pointer to a constant integer. ptr always
11        points to the same location; the integer at that location
12        cannot be modified */
13     const int *const ptr = &x;
14
15     printf( "%d\n", *ptr );
16     *ptr = 7; /* error: *ptr is const; cannot assign new value */
17     ptr = &y; /* error: ptr is const; cannot assign new address */
18     return 0; /* indicates successful termination */
19 } /* end main */
```

Fig. 7.14 | Attempting to modify a constant pointer to constant data. (Part 1 of 2.)



```
Compiling...
FIG07_14.c
c:\examples\ch07\FIG07_14.c(17) : error C2166: l-value specifies const object
c:\examples\ch07\FIG07_14.c(18) : error C2166: l-value specifies const object
Error executing cl.exe.

FIG07_12.exe - 2 error(s), 0 warning(s)
```

Fig. 7.14 | Attempting to modify a constant pointer to constant data. (Part 2 of 2.)



7.6 Bubble Sort Using Call-by-Reference

- ▶ Let's improve the bubble sort program of Fig. 6.15 to use two functions—`bubbleSort` and `swap`.
- ▶ Function `bubbleSort` sorts the array.
- ▶ It calls function `swap` (line 51) to exchange the array elements `array[j]` and `array[j + 1]` (see Fig.).
- ▶ Remember that C enforces information hiding between functions, so `swap` does not have access to individual array elements in `bubbleSort`.
- ▶ Because `bubbleSort` wants `swap` to have access to the array elements to be swapped, `bubbleSort` passes each of these elements call-by-reference to `swap`—the address of each array element is passed explicitly.

7.6 Bubble Sort Using Call-by-Reference (Cont.)



- ▶ Although entire arrays are automatically passed by reference, individual array elements are scalars and are ordinarily passed by value.
- ▶ Therefore, `bubbleSort` uses the address operator (&) on each of the array elements in the `swap` call (line 51) to effect call-by-reference as follows
 - `swap( &array[ j ], &array[ j + 1 ] );`
- ▶ Function `swap` receives `&array[j]` in pointer variable `element1Ptr` (line 59).

7.6 Bubble Sort Using Call-by-Reference (Cont.)



- ▶ Even though `swap`—because of information hiding—is not allowed to know the name `array[j]`, `swap` may use `*element1Ptr` as a synonym for `array[j]`—when `swap` references `*element1Ptr`, it's actually referencing `array[j]` in `bubbleSort`.
- ▶ Similarly, when `swap` references `*element2Ptr`, it's actually referencing `array[j + 1]` in `bubbleSort`.
- ▶ Even though `swap` is not allowed to say
 - `hold = array[ j ];`
 `array[ j ] = array[ j + 1 ];`
 `array[ j + 1 ] = hold;`

precisely the same effect is achieved by lines 61 through 63



```
1  /* Fig. 7.15: fig07_15.c
2     This program puts values into an array, sorts the values into
3     ascending order, and prints the resulting array. */
4  #include <stdio.h>
5  #define SIZE 10
6
7  void bubbleSort( int * const array, const int size ); /* prototype */
8
9  int main( void )
10 {
11     /* initialize array a */
12     int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
13
14     int i; /* counter */
15
16     printf( "Data items in original order\n" );
17
18     /* loop through array a */
19     for ( i = 0; i < SIZE; i++ ) {
20         printf( "%4d", a[ i ] );
21     } /* end for */
22
23     bubbleSort( a, SIZE ); /* sort the array */
```

Fig. 7.15 | Bubble sort with call-by-reference. (Part I of 4.)



```
24
25     printf( "\nData items in ascending order\n" );
26
27     /* loop through array a */
28     for ( i = 0; i < SIZE; i++ ) {
29         printf( "%4d", a[ i ] );
30     } /* end for */
31
32     printf( "\n" );
33     return 0; /* indicates successful termination */
34 } /* end main */
35
```

Fig. 7.15 | Bubble sort with call-by-reference. (Part 2 of 4.)



```
36  /* sort an array of integers using bubble sort algorithm */
37  void bubbleSort( int * const array, const int size )
38  {
39      void swap( int *element1Ptr, int *element2Ptr ); /* prototype */
40      int pass; /* pass counter */
41      int j; /* comparison counter */
42
43      /* loop to control passes */
44      for ( pass = 0; pass < size - 1; pass++ ) {
45
46          /* loop to control comparisons during each pass */
47          for ( j = 0; j < size - 1; j++ ) {
48
49              /* swap adjacent elements if they are out of order */
50              if ( array[ j ] > array[ j + 1 ] ) {
51                  swap( &array[ j ], &array[ j + 1 ] );
52              } /* end if */
53          } /* end inner for */
54      } /* end outer for */
55  } /* end function bubbleSort */
56
```

Fig. 7.15 | Bubble sort with call-by-reference. (Part 3 of 4.)



```
57  /* swap values at memory locations to which element1Ptr and
58     element2Ptr point */
59  void swap( int *element1Ptr, int *element2Ptr )
60  {
61      int hold = *element1Ptr;
62      *element1Ptr = *element2Ptr;
63      *element2Ptr = hold;
64  } /* end function swap */
```

Data items in original order
2 6 4 8 10 12 89 68 45 37
Data items in ascending order
2 4 6 8 10 12 37 45 68 89

Fig. 7.15 | Bubble sort with call-by-reference. (Part 4 of 4.)

7.6 Bubble Sort Using Call-by-Reference (Cont.)



- ▶ Several features of function `bubbleSort` should be noted.
- ▶ The function header (line 37) declares `array` as `int * const array` rather than `int array[]` to indicate that `bubbleSort` receives a single-subscripted array as an argument (again, these notations are interchangeable).
- ▶ Parameter `size` is declared `const` to enforce the principle of least privilege.
- ▶ Although parameter `size` receives a copy of a value in `main`, and modifying the copy cannot change the value in `main`, `bubbleSort` does not need to alter `size` to accomplish its task.

7.6 Bubble Sort Using Call-by-Reference (Cont.)



- ▶ The size of the array remains fixed during the execution of function `bubbleSort`.
- ▶ Therefore, `size` is declared `const` to ensure that it's not modified.
- ▶ If the size of the array is modified during the sorting process, the sorting algorithm might not run correctly.
- ▶ The prototype for function `swap` (line 39) is included in the body of function `bubbleSort` because `bubbleSort` is the only function that calls `swap`.

7.6 Bubble Sort Using Call-by-Reference (Cont.)



- ▶ Placing the prototype in `bubbleSort` restricts proper calls of `swap` to those made from `bubbleSort`.
- ▶ Other functions that attempt to call `swap` do not have access to a proper function prototype, so the compiler generates one automatically.
- ▶ This normally results in a prototype that does not match the function header (and generates a compilation warning or error) because the compiler assumes `int` for the return type and the parameter types.



Software Engineering Observation 7.3

Placing function prototypes in the definitions of other functions enforces the principle of least privilege by restricting proper function calls to the functions in which the prototypes appear.

7.6 Bubble Sort Using Call-by-Reference (Cont.)



- ▶ Function `bubbleSort` receives the size of the array as a parameter (line 37).
- ▶ The function must know the size of the array to sort the array.
- ▶ When an array is passed to a function, the memory address of the first element of the array is received by the function.
- ▶ The address, of course, does not convey the number of elements in the array.
- ▶ Therefore, you must pass to the function the array size.
- ▶ *[Note: Another common practice is to pass a pointer to the beginning of the array and a pointer to the location just beyond the end of the array.]*

7.6 Bubble Sort Using Call-by-Reference (Cont.)



- ▶ The difference of the two pointers is the length of the array and the resulting code is simpler.]
- ▶ In the program, the size of the array is explicitly passed to function `bubbleSort`.
- ▶ There are two main benefits to this approach—software reusability and proper software engineering.
- ▶ By defining the function to receive the array size as an argument, we enable the function to be used by any program that sorts single-subscripted integer arrays of any size.



Software Engineering Observation 7.4

When passing an array to a function, also pass the size of the array. This helps make the function reusable in many programs.

7.6 Bubble Sort Using Call-by-Reference (Cont.)



- ▶ We could have stored the array's size in a global variable that is accessible to the entire program.
- ▶ This would be more efficient, because a copy of the size is not made to pass to the function.
- ▶ However, other programs that require an integer array-sorting capability may not have the same global variable, so the function cannot be used in those programs.



Software Engineering Observation 7.5

Global variables usually violate the principle of least privilege and can lead to poor software engineering. Global variables should be used only to represent truly shared resources, such as the time of day.

7.6 Bubble Sort Using Call-by-Reference (Cont.)



- ▶ The size of the array could have been programmed directly into the function.
- ▶ This restricts the use of the function to an array of a specific size and significantly reduces its reusability.
- ▶ Only programs processing single-subscripted integer arrays of the specific size coded into the function can use the function.



7.7 `sizeof` Operator

- ▶ C provides the special unary operator `sizeof` to determine the size in bytes of an array (or any other data type) during program compilation.
- ▶ When applied to the name of an array as in Fig. 7.16 (line 14), the `sizeof` operator returns the total number of bytes in the array as an integer.
- ▶ Variables of type `float` are normally stored in 4 bytes of memory, and `array` is defined to have 20 elements.
- ▶ Therefore, there are a total of 80 bytes in `array`.



Performance Tip 7.2

sizeof is a compile-time operator, so it does not incur any execution-time overhead.



```
1  /* Fig. 7.16: fig07_16.c
2     Applying sizeof to an array name returns
3     the number of bytes in the array. */
4  #include <stdio.h>
5
6  size_t getSize( float *ptr ); /* prototype */
7
8  int main( void )
9  {
10     float array[ 20 ]; /* create array */
11
12     printf( "The number of bytes in the array is %d"
13            "\nThe number of bytes returned by getSize is %d\n",
14            sizeof( array ), getSize( array ) );
15     return 0; /* indicates successful termination */
16 } /* end main */
17
18 /* return size of ptr */
19 size_t getSize( float *ptr )
20 {
21     return sizeof( ptr );
22 } /* end function getSize */
```

Fig. 7.16 | Applying sizeof to an array name returns the number of bytes in the array. (Part I of 2.)



```
The number of bytes in the array is 80  
The number of bytes returned by getSize is 4
```

Fig. 7.16 | Applying `sizeof` to an array name returns the number of bytes in the array. (Part 2 of 2.)



7.7 sizeof Operator (Cont.)

- ▶ The number of elements in an array also can be determined with `sizeof`.
- ▶ For example, consider the following array definition:
 - `double real[ 22 ];`
- ▶ Variables of type `double` normally are stored in 8 bytes of memory.
- ▶ Thus, array `real` contains a total of 176 bytes.
- ▶ To determine the number of elements in the array, the following expression can be used:
 - `sizeof( real ) / sizeof( real[ 0 ] )`



7.7 sizeof Operator (Cont.)

- ▶ The expression determines the number of bytes in array `real` and divides that value by the number of bytes used in memory to store the first element of array `real` (a `double` value).
- ▶ Function `getSize` returns type `size_t`.
- ▶ Type `size_t` is a type defined by the C standard as the integral type (`unsigned` or `unsigned long`) of the value returned by operator `sizeof`.
- ▶ Type `size_t` is defined in header `<stddef.h>` (which is included by several headers, such as `<stdio.h>`).
- ▶ *[Note: If you attempt to compile Fig. 7.16 and receive errors, simply include `<stddef.h>` in your program.]*



7.7 sizeof Operator (Cont.)

- ▶ Figure 7.17 calculates the number of bytes used to store each of the standard data types.
- ▶ The results could be different between computers.



Portability Tip 7.2

The number of bytes used to store a particular data type may vary between systems. When writing programs that depend on data type sizes and that will run on several computer systems, use `sizeof` to determine the number of bytes used to store the data types.



```
1  /* Fig. 7.17: fig07_17.c
2     Demonstrating the sizeof operator */
3  #include <stdio.h>
4
5  int main( void )
6  {
7     char c;
8     short s;
9     int i;
10    long l;
11    float f;
12    double d;
13    long double ld;
14    int array[ 20 ]; /* create array of 20 int elements */
15    int *ptr = array; /* create pointer to array */
16
17    printf( "    sizeof c = %d\\tsizeof(char) = %d"
18           "\\n    sizeof s = %d\\tsizeof(short) = %d"
19           "\\n    sizeof i = %d\\tsizeof(int) = %d"
20           "\\n    sizeof l = %d\\tsizeof(long) = %d"
21           "\\n    sizeof f = %d\\tsizeof(float) = %d"
22           "\\n    sizeof d = %d\\tsizeof(double) = %d"
```

Fig. 7.17 | Using operator `sizeof` to determine standard data type sizes. (Part I of 2.)



```
23         "\n    sizeof ld = %d\\tsizeof(long double) = %d"
24         "\n sizeof array = %d"
25         "\n    sizeof ptr = %d\\n",
26         sizeof c, sizeof( char ), sizeof s, sizeof( short ), sizeof i,
27         sizeof( int ), sizeof l, sizeof( long ), sizeof f,
28         sizeof( float ), sizeof d, sizeof( double ), sizeof ld,
29         sizeof( long double ), sizeof array, sizeof ptr );
30     return 0; /* indicates successful termination */
31 } /* end main */
```

sizeof c = 1	sizeof(char) = 1
sizeof s = 2	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	

Fig. 7.17 | Using operator sizeof to determine standard data type sizes. (Part 2 of 2.)



7.7 sizeof Operator (Cont.)

- ▶ Operator `sizeof` can be applied to any variable name, type or value (including the value of an expression).
- ▶ When applied to a variable name (that is not an array name) or a constant, the number of bytes used to store the specific type of variable or constant is returned.
- ▶ The parentheses used with `sizeof` are required if a type name with two words is supplied as its operand (such as `long double` or `unsigned short`).
- ▶ Omitting the parentheses in this case results in a syntax error.
- ▶ The parentheses are not required if a variable name or a one-word type name is supplied as its operand, but they can still be included without causing an error.



7.8 Pointer Expressions and Pointer Arithmetic

- ▶ Pointers are valid operands in arithmetic expressions, assignment expressions and comparison expressions.
- ▶ However, not all the operators normally used in these expressions are valid in conjunction with pointer variables.
- ▶ This section describes the operators that can have pointers as operands, and how these operators are used.
- ▶ A limited set of arithmetic operations may be performed on pointers.
- ▶ A pointer may be incremented (++) or decremented (--), an integer may be added to a pointer (+ or +=), an integer may be subtracted from a pointer (- or -=) and one pointer may be subtracted from another.



7.8 Pointer Expressions and Pointer Arithmetic (Cont.)

- ▶ Assume that array `int v[5]` has been defined and its first element is at location 3000 in memory.
- ▶ Assume pointer `vPtr` has been initialized to point to `v[0]`—i.e., the value of `vPtr` is 3000.
- ▶ Figure 7.18 illustrates this situation for a machine with 4-byte integers.
- ▶ Variable `vPtr` can be initialized to point to array `v` with either of the statements



Portability Tip 7.3

Most computers today have 2-byte or 4-byte integers. Some of the newer machines use 8-byte integers. Because the results of pointer arithmetic depend on the size of the objects a pointer points to, pointer arithmetic is machine dependent.

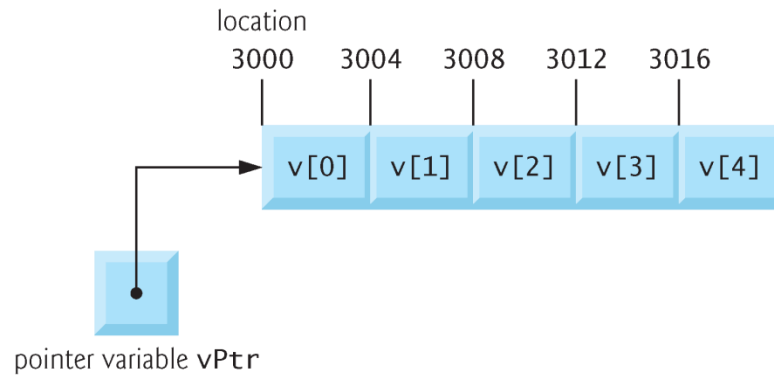


Fig. 7.18 | Array `v` and a pointer variable `vPtr` that points to `v`.



7.8 Pointer Expressions and Pointer Arithmetic (Cont.)

- ▶ In conventional arithmetic, $3000 + 2$ yields the value 3002.
- ▶ This is normally not the case with pointer arithmetic.
- ▶ When an integer is added to or subtracted from a pointer, the pointer is not incremented or decremented simply by that integer, but by that integer times the size of the object to which the pointer refers.
- ▶ The number of bytes depends on the object's data type.
- ▶ For example, the statement
 - `vPtr += 2;`would produce 3008 ($3000 + 2 * 4$), assuming an integer is stored in 4 bytes of memory.



7.8 Pointer Expressions and Pointer Arithmetic (Cont.)

- ▶ In the array `v`, `vPtr` would now point to `v[2]` (Fig. 7.19).
- ▶ If an integer is stored in 2 bytes of memory, then the preceding calculation would result in memory location 3004 ($3000 + 2 * 2$).
- ▶ If the array were of a different data type, the preceding statement would increment the pointer by twice the number of bytes that it takes to store an object of that data type.
- ▶ When performing pointer arithmetic on a character array, the results will be consistent with regular arithmetic, because each character is 1 byte long.

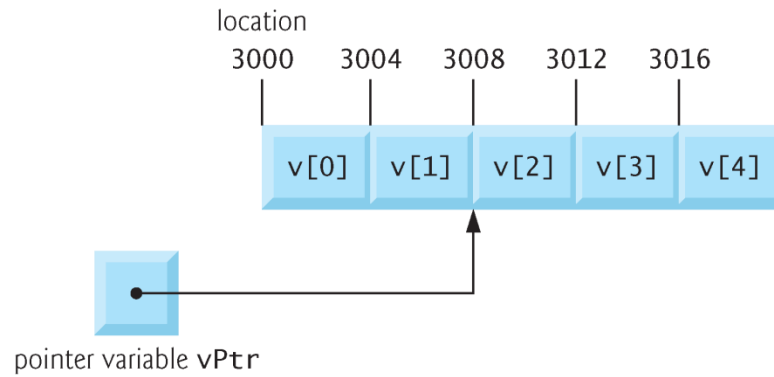


Fig. 7.19 | The pointer `vPtr` after pointer arithmetic.



7.8 Pointer Expressions and Pointer Arithmetic (Cont.)

- ▶ If `vPtr` had been incremented to 3016, which points to `v[4]`, the statement
 - `vPtr -= 4;`would set `vPtr` back to 3000—the beginning of the array.
- ▶ If a pointer is being incremented or decremented by one, the increment (`++`) and decrement (`--`) operators can be used.
- ▶ Either of the statements
 - `++vPtr;`
`vPtr++;`increments the pointer to point to the next location in the array.



7.8 Pointer Expressions and Pointer Arithmetic (Cont.)

- ▶ Either of the statements

- `--vPtr;`
`vPtr--;`

decrements the pointer to point to the previous element of the array.

- ▶ Pointer variables may be subtracted from one another.



7.8 Pointer Expressions and Pointer Arithmetic (Cont.)

- ▶ For example, if `vPtr` contains the location 3000, and `v2Ptr` contains the address 3008, the statement
 - `x = v2Ptr - vPtr;`would assign to `x` the number of array elements from `vPtr` to `v2Ptr`, in this case 2 (not 8).
- ▶ Pointer arithmetic is meaningless unless performed on an array.
- ▶ We cannot assume that two variables of the same type are stored contiguously in memory unless they're adjacent elements of an array.



Common Programming Error 7.5

Using pointer arithmetic on a pointer that does not refer to an element in an array.



Common Programming Error 7.6

Subtracting or comparing two pointers that do not refer to elements in the same array.



Common Programming Error 7.7

Running off either end of an array when using pointer arithmetic.



7.8 Pointer Expressions and Pointer Arithmetic (Cont.)

- ▶ A pointer can be assigned to another pointer if both have the same type.
- ▶ The exception to this rule is the **pointer to void** (i.e., `void *`), which is a generic pointer that can represent any pointer type.
- ▶ All pointer types can be assigned a pointer to `void`, and a pointer to `void` can be assigned a pointer of any type.
- ▶ In both cases, a cast operation is not required.
- ▶ A pointer to `void` cannot be dereferenced.



7.8 Pointer Expressions and Pointer Arithmetic (Cont.)

- ▶ Consider this: The compiler knows that a pointer to `int` refers to 4 bytes of memory on a machine with 4-byte integers, but a pointer to `void` simply contains a memory location for an unknown data type—the precise number of bytes to which the pointer refers is not known by the compiler.
- ▶ The compiler must know the data type to determine the number of bytes to be dereferenced for a particular pointer.



Common Programming Error 7.8

Assigning a pointer of one type to a pointer of another type if neither is of type `void *` is a syntax error.



Common Programming Error 7.9

Dereferencing a `void *` pointer is a syntax error.



7.8 Pointer Expressions and Pointer Arithmetic (Cont.)

- ▶ Pointers can be compared using equality and relational operators, but such comparisons are meaningless unless the pointers point to elements of the same array.
- ▶ Pointer comparisons compare the addresses stored in the pointers.
- ▶ A comparison of two pointers pointing to elements in the same array could show, for example, that one pointer points to a higher-numbered element of the array than the other pointer does.
- ▶ A common use of pointer comparison is determining whether a pointer is **NULL**.



7.9 Relationship between Pointers and Arrays

- ▶ Arrays and pointers are intimately related in C and often may be used interchangeably.
- ▶ An array name can be thought of as a constant pointer.
- ▶ Pointers can be used to do any operation involving array subscripting.
- ▶ Assume that integer array `b[5]` and integer pointer variable `bPtr` have been defined.
- ▶ Since the array name (without a subscript) is a pointer to the first element of the array, we can set `bPtr` equal to the address of the first element in array `b` with the statement
 - `bPtr = b;`



7.9 Relationship between Pointers and Arrays (Cont.)

- ▶ This statement is equivalent to taking the address of the array's first element as follows:
 - `bPtr = &b[ 0 ];`
- ▶ Array element `b[3]` can alternatively be referenced with the pointer expression
 - `*( bPtr + 3 )`
- ▶ The 3 in the above expression is the **offset** to the pointer.
- ▶ When the pointer points to the beginning of an array, the offset indicates which element of the array should be referenced, and the offset value is identical to the array subscript.
- ▶ The preceding notation is referred to as **pointer/offset notation**.



7.9 Relationship between Pointers and Arrays (Cont.)

- ▶ The parentheses are necessary because the precedence of `*` is higher than the precedence of `+`.
- ▶ Without the parentheses, the above expression would add 3 to the value of the expression `*bPtr` (i.e., 3 would be added to `b[0]`, assuming `bPtr` points to the beginning of the array).
- ▶ Just as the array element can be referenced with a pointer expression, the address
 - `&b[ 3 ]`can be written with the pointer expression
 - `bPtr + 3`
- ▶ The array itself can be treated as a pointer and used in pointer arithmetic.



7.9 Relationship between Pointers and Arrays (Cont.)

- ▶ For example, the expression
 - `*( b + 3 )`also refers to the array element `b[3]`.
- ▶ In general, all subscripted array expressions can be written with a pointer and an offset.
- ▶ In this case, pointer/offset notation was used with the name of the array as a pointer.
- ▶ The preceding statement does not modify the array name in any way; `b` still points to the first element in the array.
- ▶ Pointers can be subscripted exactly as arrays can.



7.9 Relationship between Pointers and Arrays (Cont.)

- ▶ For example, if `bPtr` has the value `b`, the expression
 - `bPtr[ 1 ]`refers to the array element `b[1]`.
- ▶ This is referred to as **pointer/subscript notation**.
- ▶ Remember that an array name is essentially a constant pointer; it always points to the beginning of the array.
- ▶ Thus, the expression
 - `b += 3`is invalid because it attempts to modify the value of the array name with pointer arithmetic.



Common Programming Error 7.10

Attempting to modify an array name with pointer arithmetic is a syntax error.

7.9 Relationship between Pointers and Arrays (Cont.)



- ▶ Figure 7.20 uses the four methods we have discussed for referring to array elements—array subscripting, pointer/offset with the array name as a pointer, **pointer subscripting**, and pointer/offset with a pointer—to print the four elements of the integer array `b`.



```
1  /* Fig. 7.20: fig07_20.cpp
2     Using subscripting and pointer notations with arrays */
3
4  #include <stdio.h>
5
6  int main( void )
7  {
8      int b[] = { 10, 20, 30, 40 }; /* initialize array b */
9      int *bPtr = b; /* set bPtr to point to array b */
10     int i; /* counter */
11     int offset; /* counter */
12
13     /* output array b using array subscript notation */
14     printf( "Array b printed with:\nArray subscript notation\n" );
15
16     /* loop through array b */
17     for ( i = 0; i < 4; i++ ) {
18         printf( "b[ %d ] = %d\n", i, b[ i ] );
19     } /* end for */
20
21     /* output array b using array name and pointer/offset notation */
22     printf( "\nPointer/offset notation where\n"
23         "the pointer is the array name\n" );
```

Fig. 7.20 | Using four methods of referencing array elements. (Part I of 3.)



```
24
25  /* loop through array b */
26  for ( offset = 0; offset < 4; offset++ ) {
27      printf( "( b + %d ) = %d\n", offset, *( b + offset ) );
28  } /* end for */
29
30  /* output array b using bPtr and array subscript notation */
31  printf( "\nPointer subscript notation\n" );
32
33  /* loop through array b */
34  for ( i = 0; i < 4; i++ ) {
35      printf( "bPtr[ %d ] = %d\n", i, bPtr[ i ] );
36  } /* end for */
37
38  /* output array b using bPtr and pointer/offset notation */
39  printf( "\nPointer/offset notation\n" );
40
41  /* loop through array b */
42  for ( offset = 0; offset < 4; offset++ ) {
43      printf( "( bPtr + %d ) = %d\n", offset, *( bPtr + offset ) );
44  } /* end for */
45
46  return 0; /* indicates successful termination */
47 } /* end main */
```

Fig. 7.20 | Using four methods of referencing array elements. (Part 2 of 3.)



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
```

Fig. 7.20 | Using four methods of referencing array elements. (Part 3 of 3.)



7.9 Relationship between Pointers and Arrays (Cont.)

- ▶ To further illustrate the interchangeability of arrays and pointers, let's look at the two string-copying functions—`copy1` and `copy2`—in the program of Fig. 7.21.
- ▶ Both functions copy a string (possibly a character array) into a character array.
- ▶ After a comparison of the function prototypes for `copy1` and `copy2`, the functions appear identical.
- ▶ They accomplish the same task; however, they're implemented differently.



```
1  /* Fig. 7.21: fig07_21.c
2     Copying a string using array notation and pointer notation. */
3  #include <stdio.h>
4
5  void copy1( char * const s1, const char * const s2 ); /* prototype */
6  void copy2( char *s1, const char *s2 ); /* prototype */
7
8  int main( void )
9  {
10     char string1[ 10 ]; /* create array string1 */
11     char *string2 = "Hello"; /* create a pointer to a string */
12     char string3[ 10 ]; /* create array string3 */
13     char string4[] = "Good Bye"; /* create a pointer to a string */
14
15     copy1( string1, string2 );
16     printf( "string1 = %s\n", string1 );
17
18     copy2( string3, string4 );
19     printf( "string3 = %s\n", string3 );
20     return 0; /* indicates successful termination */
21 }
```

Fig. 7.21 | Copying a string using array notation and pointer notation. (Part I of 2.)



```
22
23  /* copy s2 to s1 using array notation */
24 void copy1( char * const s1, const char * const s2 )
25 {
26     int i; /* counter */
27
28     /* loop through strings */
29     for ( i = 0; ( s1[ i ] = s2[ i ] ) != '\0'; i++ ) {
30         ; /* do nothing in body */
31     } /* end for */
32 } /* end function copy1 */
33
34 /* copy s2 to s1 using pointer notation */
35 void copy2( char *s1, const char *s2 )
36 {
37     /* loop through strings */
38     for ( ; ( *s1 = *s2 ) != '\0'; s1++, s2++ ) {
39         ; /* do nothing in body */
40     } /* end for */
41 } /* end function copy2 */
```

```
string1 = Hello
string3 = Good Bye
```

Fig. 7.21 | Copying a string using array notation and pointer notation. (Part 2 of 2.)



7.9 Relationship between Pointers and Arrays (Cont.)

- ▶ Function `copy1` uses array subscript notation to copy the string in `s2` to the character array `s1`.
- ▶ The function defines counter variable `i` as the array subscript.
- ▶ The `for` statement header (line 29) performs the entire copy operation—its body is the empty statement.
- ▶ The header specifies that `i` is initialized to zero and incremented by one on each iteration of the loop.
- ▶ The expression `s1[i] = s2[i]` copies one character from `s2` to `s1`.
- ▶ When the null character is encountered in `s2`, it's assigned to `s1`, and the value of the assignment becomes the value assigned to the left operand (`s1`).



7.9 Relationship between Pointers and Arrays (Cont.)

- ▶ The loop terminates because the integer value of the null character is zero (false).
- ▶ Function `copy2` uses pointers and pointer arithmetic to copy the string in `s2` to the character array `s1`.
- ▶ Again, the `for` statement header (line 38) performs the entire copy operation.
- ▶ The header does not include any variable initialization.
- ▶ As in function `copy1`, the expression `(*s1 = *s2)` performs the copy operation.
- ▶ Pointer `s2` is dereferenced, and the resulting character is assigned to the dereferenced pointer `*s1`.



7.9 Relationship between Pointers and Arrays (Cont.)

- ▶ After the assignment in the condition, the pointers are incremented to point to the next element of array `s1` and the next character of string `s2`, respectively.
- ▶ When the null character is encountered in `s2`, it's assigned to the dereferenced pointer `s1` and the loop terminates.
- ▶ The first argument to both `copy1` and `copy2` must be an array large enough to hold the string in the second argument.
- ▶ Otherwise, an error may occur when an attempt is made to write into a memory location that is not part of the array.
- ▶ Also, the second parameter of each function is declared as `const char *` (a constant string).



7.9 Relationship between Pointers and Arrays (Cont.)

- ▶ In both functions, the second argument is copied into the first argument—characters are read from it one at a time, but the characters are never modified.
- ▶ Therefore, the second parameter is declared to point to a constant value so that the principle of least privilege is enforced—neither function requires the capability of modifying the second argument, so neither function is provided with that capability.



7.10 Arrays of Pointers

- ▶ Arrays may contain pointers.
- ▶ A common use of an **array of pointers** is to form an **array of strings**, referred to simply as a **string array**.
- ▶ Each entry in the array is a string, but in C a string is essentially a pointer to its first character.
- ▶ So each entry in an array of strings is actually a pointer to the first character of a string.
- ▶ Consider the definition of string array `suit`, which might be useful in representing a deck of cards.
 - `const char *suit[ 4 ] = { "Hearts", "Diamonds", "Clubs", "Spades" };`



7.10 Arrays of Pointers (Cont.)

- ▶ The `suit[4]` portion of the definition indicates an array of 4 elements.
- ▶ The `char *` portion of the declaration indicates that each element of array `suit` is of type “pointer to `char`.”
- ▶ Qualifier `const` indicates that the strings pointed to by each element pointer will not be modified.
- ▶ The four values to be placed in the array are "Hearts", "Diamonds", "Clubs" and "Spades".
- ▶ Each is stored in memory as a null-terminated character string that is one character longer than the number of characters between quotes.



7.10 Arrays of Pointers (Cont.)

- ▶ The four strings are 7, 9, 6 and 7 characters long, respectively.
- ▶ Although it appears as though these strings are being placed in the `suit` array, only pointers are actually stored in the array (Fig. 7.22).
- ▶ Each pointer points to the first character of its corresponding string.
- ▶ Thus, even though the `suit` array is fixed in size, it provides access to character strings of any length.
- ▶ This flexibility is one example of C's powerful data-structuring capabilities.

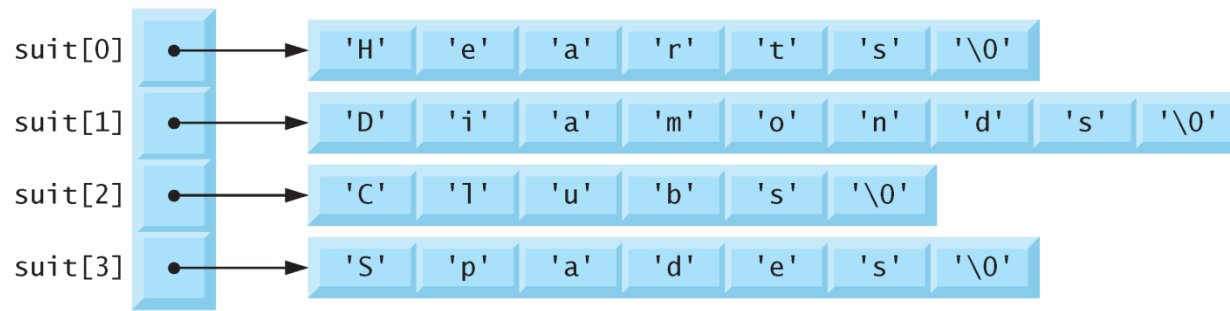


Fig. 7.22 | Graphical representation of the `suit` array.



7.10 Arrays of Pointers (Cont.)

- ▶ The suits could have been placed in a two-dimensional array, in which each row would represent a suit and each column would represent a letter from a suit name.
- ▶ Such a data structure would have to have a fixed number of columns per row, and that number would have to be as large as the largest string.
- ▶ Therefore, considerable memory could be wasted when a large number of strings were being stored with most strings shorter than the longest string.



7.11 Case Study: Card Shuffling and Dealing Simulation

- ▶ In this section, we use random number generation to develop a card shuffling and dealing simulation program.
- ▶ This program can then be used to implement programs that play specific card games.
- ▶ To reveal some subtle performance problems, we have intentionally used suboptimal shuffling and dealing algorithms.
- ▶ In this chapter's exercises and in Chapter 10, we develop more efficient algorithms.
- ▶ Using the top-down, stepwise refinement approach, we develop a program that will shuffle a deck of 52 playing cards and then deal each of the 52 cards.



7.11 Case Study: Card Shuffling and Dealing Simulation (Cont.)

- ▶ The top-down approach is particularly useful in attacking larger, more complex problems than we have seen.
- ▶ We use 4-by-13 double-subscripted array **deck** to represent the deck of playing cards (Fig. 7.23).
- ▶ The rows correspond to the suits—row 0 corresponds to hearts, row 1 to diamonds, row 2 to clubs and row 3 to spades.
- ▶ The columns correspond to the face values of the cards—0 through 9 correspond to ace through ten, and columns 10 through 12 correspond to jack, queen and king.
- ▶ We shall load string array **suit** with character strings representing the four suits, and string array **face** with character strings representing the thirteen face values.

		Ace	Two	Three	Four	Five	Six	Seven	Eight	Nine	Ten	Jack	Queen	King
		0	1	2	3	4	5	6	7	8	9	10	11	12
Hearts	0													
Diamonds	1													
Clubs	2													
Spades	3													

deck[2][12] represents the King of Clubs

Clubs King

Fig. 7.23 | Double-subscripted array representation of a deck of cards.



7.11 Case Study: Card Shuffling and Dealing Simulation (Cont.)

- ▶ This simulated deck of cards may be shuffled as follows.
- ▶ First the array `deck` is cleared to zeros.
- ▶ Then, a `row` (0–3) and a `column` (0–12) are each chosen at random.
- ▶ The number 1 is inserted in array element `deck[row][column]` to indicate that this card is going to be the first one dealt from the shuffled deck.
- ▶ This process continues with the numbers 2, 3, ..., 52 being randomly inserted in the `deck` array to indicate which cards are to be placed second, third, ..., and fifty-second in the shuffled deck.



7.11 Case Study: Card Shuffling and Dealing Simulation (Cont.)

- ▶ As the **deck** array begins to fill with card numbers, it's possible that a card will be selected twice—i.e., **deck[row] [column]** will be nonzero when it's selected.
- ▶ This selection is simply ignored and other **rows** and **columns** are repeatedly chosen at random until an unselected card is found.
- ▶ Eventually, the numbers 1 through 52 will occupy the 52 slots of the **deck** array.
- ▶ At this point, the deck of cards is fully shuffled.



7.11 Case Study: Card Shuffling and Dealing Simulation (Cont.)

- ▶ This shuffling algorithm could execute indefinitely if cards that have already been shuffled are repeatedly selected at random.
- ▶ This phenomenon is known as **indefinite postponement**.
- ▶ In the exercises, we discuss a better shuffling algorithm that eliminates the possibility of indefinite postponement.



Performance Tip 7.3

Sometimes an algorithm that emerges in a “natural” way can contain subtle performance problems, such as indefinite postponement. Seek algorithms that avoid indefinite postponement.



7.11 Case Study: Card Shuffling and Dealing Simulation (Cont.)

- ▶ To deal the first card, we search the array for `deck[row][column]` equal to 1.
- ▶ This is accomplished with a nested `for` statement that varies `row` from 0 to 3 and `column` from 0 to 12.
- ▶ What card does that element of the array correspond to?
- ▶ The `suit` array has been preloaded with the four suits, so to get the suit, we print the character string `suit[row]`.
- ▶ Similarly, to get the face value of the card, we print the character string `face[column]`.
- ▶ We also print the character string " of ".



7.11 Case Study: Card Shuffling and Dealing Simulation (Cont.)

- ▶ Printing this information in the proper order enables us to print each card in the form "King of Clubs", "Ace of Diamonds" and so on.
- ▶ Let's proceed with the top-down, stepwise refinement process.
- ▶ The top is simply
 - *Shuffle and deal 52 cards*
- ▶ Our first refinement yields:
 - Initialize the suit array
 - Initialize the face array
 - Initialize the deck array
 - Shuffle the deck
 - Deal 52 cards



7.11 Case Study: Card Shuffling and Dealing Simulation (Cont.)

- ▶ “Shuffle the deck” may be expanded as follows:
 - For each of the 52 cards
Place card number in randomly selected unoccupied slot of deck
- ▶ “Deal 52 cards” may be expanded as follows:
 - For each of the 52 cards
Find card number in deck array and print face and suit of card



7.11 Case Study: Card Shuffling and Dealing Simulation (Cont.)

- ▶ Incorporating these expansions yields our complete second refinement:
 - Initialize the suit array
 - Initialize the face array
 - Initialize the deck array

For each of the 52 cards

Place card 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



7.11 Case Study: Card Shuffling and Dealing Simulation (Cont.)

- ▶ “Place card number in randomly selected unoccupied slot of deck” may be expanded as:
 - 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

- ▶ “Find card number in deck array and print face and suit of card” may be expanded as:
 - For each slot of the deck array
If slot contains card number
Print the face and suit of the card



7.11 Case Study: Card Shuffling and Dealing Simulation (Cont.)

- ▶ Incorporating these expansions yields our third refinement:

- Initialize the suit array
Initialize the face array
Initialize the deck array

For each of the 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 slot of deck array
If slot contains desired card number
Print the face and suit of the card



7.11 Case Study: Card Shuffling and Dealing Simulation (Cont.)

- ▶ This completes the refinement process.
- ▶ This program is more efficient if the shuffle and deal portions of the algorithm are combined so that each card is dealt as it's placed in the deck.
- ▶ We have chosen to program these operations separately because normally cards are dealt after they're shuffled (not while they're shuffled).



7.11 Case Study: Card Shuffling and Dealing Simulation (Cont.)

- ▶ The card shuffling and dealing program is shown in Fig. 7.24, and a sample execution is shown in Fig. 7.25.
- ▶ Conversion specifier `%s` is used to print strings of characters in the calls to `printf`.
- ▶ The corresponding argument in the `printf` call must be a pointer to `char` (or a `char` array).
- ▶ The format specification `"%5s of %-8s"` (line 73) prints a character string right justified in a field of five characters followed by `" of "` and a character string left justified in a field of eight characters.
- ▶ The minus sign in `%-8s` signifies left justification.



```
1  /* Fig. 7.24: fig07_24.c
2     Card shuffling dealing program */
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <time.h>
6
7  /* prototypes */
8  void shuffle( int wDeck[][ 13 ] );
9  void deal( const int wDeck[][ 13 ], const char *wFace[],
10             const char *wSuit[] );
11
12 int main( void )
13 {
14     /* initialize suit array */
15     const char *suit[ 4 ] = { "Hearts", "Diamonds", "Clubs", "Spades" };
16
17     /* initialize face array */
18     const char *face[ 13 ] =
19         { "Ace", "Deuce", "Three", "Four",
20           "Five", "Six", "Seven", "Eight",
21           "Nine", "Ten", "Jack", "Queen", "King" };
22 }
```

Fig. 7.24 | Card dealing program. (Part 1 of 4.)



```
23  /* initialize deck array */
24  int deck[ 4 ][ 13 ] = { 0 };
25
26  srand( time( 0 ) ); /* seed random-number generator */
27
28  shuffle( deck ); /* shuffle the deck */
29  deal( deck, face, suit ); /* deal the deck */
30  return 0; /* indicates successful termination */
31 } /* end main */
32
```

Fig. 7.24 | Card dealing program. (Part 2 of 4.)



```
33  /* shuffle cards in deck */
34  void shuffle( int wDeck[][ 13 ] )
35  {
36      int row; /* row number */
37      int column; /* column number */
38      int card; /* counter */
39
40      /* for each of the 52 cards, choose slot of deck randomly */
41      for ( card = 1; card <= 52; card++ ) {
42
43          /* choose new random location until unoccupied slot found */
44          do {
45              row = rand() % 4;
46              column = rand() % 13;
47          } while( wDeck[ row ][ column ] != 0 ); /* end do...while */
48
49          /* place card number in chosen slot of deck */
50          wDeck[ row ][ column ] = card;
51      } /* end for */
52  } /* end function shuffle */
53
```

Fig. 7.24 | Card dealing program. (Part 3 of 4.)



```
54  /* deal cards in deck */
55  void deal( const int wDeck[][ 13 ], const char *wFace[],
56            const char *wSuit[] )
57  {
58      int card; /* card counter */
59      int row; /* row counter */
60      int column; /* column counter */
61
62      /* deal each of the 52 cards */
63      for ( card = 1; card <= 52; card++ ) {
64          /* loop through rows of wDeck */
65
66          for ( row = 0; row <= 3; row++ ) {
67
68              /* loop through columns of wDeck for current row */
69              for ( column = 0; column <= 12; column++ ) {
70
71                  /* if slot contains current card, display card */
72                  if ( wDeck[ row ][ column ] == card ) {
73                      printf( "%5s of %-8s%c", wFace[ column ], wSuit[ row ],
74                          card % 2 == 0 ? '\n' : '\t' );
75                  } /* end if */
76              } /* end for */
77          } /* end for */
78      } /* end for */
79  } /* end function deal */
```



Nine of Hearts	Five of Clubs
Queen of Spades	Three of Spades
Queen of Hearts	Ace of Clubs
King of Hearts	Six of Spades
Jack of Diamonds	Five of Spades
Seven of Hearts	King of Clubs
Three of Clubs	Eight of Hearts
Three of Diamonds	Four of Diamonds
Queen of Diamonds	Five of Diamonds
Six of Diamonds	Five of Hearts
Ace of Spades	Six of Hearts
Nine of Diamonds	Queen of Clubs
Eight of Spades	Nine of Clubs
Deuce of Clubs	Six of Clubs
Deuce of Spades	Jack of Clubs
Four of Clubs	Eight of Clubs
Four of Spades	Seven of Spades
Seven of Diamonds	Seven of Clubs
King of Spades	Ten of Diamonds
Jack of Hearts	Ace of Hearts
Jack of Spades	Ten of Clubs
Eight of Diamonds	Deuce of Diamonds
Ace of Diamonds	Nine of Spades
Four of Hearts	Deuce of Hearts
King of Diamonds	Ten of Spades
Three of Hearts	Ten of Hearts

Fig. 7.25 | Sample run of card dealing program.



7.11 Case Study: Card Shuffling and Dealing Simulation (Cont.)

- ▶ There's a weakness in the dealing algorithm.
- ▶ Once a match is found, the two inner **for** statements continue searching the remaining elements of **deck** for a match.
- ▶ We correct this deficiency in this chapter's exercises and in a Chapter 10 case study.



7.12 Pointers to Functions

- ▶ A **pointer to a function** contains the address of the function in memory.
- ▶ In Chapter 6, we saw that an array name is really the address in memory of the first element of the array.
- ▶ Similarly, a function name is really the starting address in memory of the code that performs the function's task.
- ▶ Pointers to functions can be passed to functions, returned from functions, stored in arrays and assigned to other function pointers.



7.12 Pointers to Functions (Cont.)

- ▶ To illustrate the use of pointers to functions, Fig. 7.26 presents a modified version of the bubble sort program in Fig. 7.15.
- ▶ The new version consists of `main` and functions `bubble`, `swap`, `ascending` and `descending`.
- ▶ Function `bubbleSort` receives a pointer to a function—either function `ascending` or function `descending`—as an argument, in addition to an integer array and the size of the array.



7.12 Pointers to Functions (Cont.)

- ▶ The program prompts the user to choose whether the array should be sorted in ascending or in descending order.
- ▶ If the user enters **1**, a pointer to function **ascending** is passed to function **bubble**, causing the array to be sorted into increasing order.
- ▶ If the user enters **2**, a pointer to function **descending** is passed to function **bubble**, causing the array to be sorted into decreasing order.
- ▶ The output of the program is shown in Fig. 7.27.



```
1  /* Fig. 7.26: fig07_26.c
2     Multipurpose sorting program using function pointers */
3  #include <stdio.h>
4  #define SIZE 10
5
6  /* prototypes */
7  void bubble( int work[], const int size, int (*compare)( int a, int b ) );
8  int ascending( int a, int b );
9  int descending( int a, int b );
10
11 int main( void )
12 {
13     int order; /* 1 for ascending order or 2 for descending order */
14     int counter; /* counter */
15
16     /* initialize array a */
17     int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
18
19     printf( "Enter 1 to sort in ascending order,\n"
20            "Enter 2 to sort in descending order: " );
21     scanf( "%d", &order );
22
23     printf( "\nData items in original order\n" );
```

Fig. 7.26 | Multipurpose sorting program using function pointers. (Part I of 4.)



```
24
25  /* output original array */
26  for ( counter = 0; counter < SIZE; counter++ ) {
27      printf( "%5d", a[ counter ] );
28  } /* end for */
29
30  /* sort array in ascending order; pass function ascending as an
31     argument to specify ascending sorting order */
32  if ( order == 1 ) {
33      bubble( a, SIZE, ascending );
34      printf( "\nData items in ascending order\n" );
35  } /* end if */
36  else { /* pass function descending */
37      bubble( a, SIZE, descending );
38      printf( "\nData items in descending order\n" );
39  } /* end else */
40
41  /* output sorted array */
42  for ( counter = 0; counter < SIZE; counter++ ) {
43      printf( "%5d", a[ counter ] );
44  } /* end for */
45
46  printf( "\n" );
47  return 0; /* indicates successful termination */
48 } /* end main */
```

Fig. 7.26 | Multipurpose sorting program using function pointers. (Part 2 of 4.)



```
49
50  /* multipurpose bubble sort; parameter compare is a pointer to
51     the comparison function that determines sorting order */
52  void bubble( int work[], const int size, int (*compare)( int a, int b ) )
53  {
54      int pass; /* pass counter */
55      int count; /* comparison counter */
56
57      void swap( int *element1Ptr, int *element2ptr ); /* prototype */
58
59      /* loop to control passes */
60      for ( pass = 1; pass < size; pass++ ) {
61
62          /* loop to control number of comparisons per pass */
63          for ( count = 0; count < size - 1; count++ ) {
64
65              /* if adjacent elements are out of order, swap them */
66              if ( (*compare)( work[ count ], work[ count + 1 ] ) ) {
67                  swap( &work[ count ], &work[ count + 1 ] );
68              } /* end if */
69          } /* end for */
70      } /* end for */
71  } /* end function bubble */
72
```

Fig. 7.26 | Multipurpose sorting program using function pointers. (Part 3 of 4.)



```
73  /* swap values at memory locations to which element1Ptr and
74     element2Ptr point */
75  void swap( int *element1Ptr, int *element2Ptr )
76  {
77      int hold; /* temporary holding variable */
78
79      hold = *element1Ptr;
80      *element1Ptr = *element2Ptr;
81      *element2Ptr = hold;
82  } /* end function swap */
83
84  /* determine whether elements are out of order for an ascending
85     order sort */
86  int ascending( int a, int b )
87  {
88      return b < a; /* swap if b is less than a */
89  } /* end function ascending */
90
91  /* determine whether elements are out of order for a descending
92     order sort */
93  int descending( int a, int b )
94  {
95      return b > a; /* swap if b is greater than a */
96  } /* end function descending */
```

Fig. 7.26 | Multipurpose sorting program using function pointers. (Part 4 of 4.)



```
Enter 1 to sort in ascending order,  
Enter 2 to sort in descending order: 1
```

```
Data items in original order
```

```
2    6    4    8    10   12   89   68   45   37
```

```
Data items in ascending order
```

```
2    4    6    8    10   12   37   45   68   89
```

```
Enter 1 to sort in ascending order,  
Enter 2 to sort in descending order: 2
```

```
Data items in original order
```

```
2    6    4    8    10   12   89   68   45   37
```

```
Data items in descending order
```

```
89   68   45   37   12   10    8    6    4    2
```

Fig. 7.27 | The outputs of the bubble sort program in Fig. 7.26.



7.12 Pointers to Functions (Cont.)

- ▶ The following parameter appears in the function header for `bubble` (line 52)
 - `int (*compare)( int a, int b )`
- ▶ This tells `bubble` to expect a parameter (`compare`) that is a pointer to a function that receives two integer parameters and returns an integer result.



7.12 Pointers to Functions (Cont.)

- ▶ Parentheses are needed around `*compare` to group `*` with `compare` to indicate that `compare` is a pointer.
- ▶ If we had not included the parentheses, the declaration would have been

- `int *compare( int a, int b )`

which declares a function that receives two integers as parameters and returns a pointer to an integer.



7.12 Pointers to Functions (Cont.)

- ▶ The function prototype for `bubble` is shown in line 7.
- ▶ The prototype could have been written as
 - `int (*)( int, int );`
- ▶ without the function-pointer name and parameter names.
- ▶ The function passed to `bubble` is called in an `if` statement (line 66) as follows:
 - `if ((*compare)( work[count], work[count + 1]))`
- ▶ Just as a pointer to a variable is dereferenced to access the value of the variable, a pointer to a function is dereferenced to use the function.



7.12 Pointers to Functions (Cont.)

- ▶ The call to the function could have been made without dereferencing the pointer as in
 - `if (compare(work[count], work[count + 1]))`
- ▶ which uses the pointer directly as the function name.
- ▶ We prefer the first method of calling a function through a pointer because it explicitly illustrates that `compare` is a pointer to a function that is dereferenced to call the function.
- ▶ The second method of calling a function through a pointer makes it appear as though `compare` is an actual function.
- ▶ This may be confusing to a user of the program who would like to see the definition of function `compare` and finds that it's never defined in the file.



7.12 Pointers to Functions (Cont.)

- ▶ A common use of **function pointers** is in text-based menu-driven systems.
- ▶ A user is prompted to select an option from a menu (possibly from 1 to 5) by typing the menu item's number.
- ▶ Each option is serviced by a different function.
- ▶ Pointers to each function are stored in an array of pointers to functions.
- ▶ The user's choice is used as a subscript in the array, and the pointer in the array is used to call the function.



7.12 Pointers to Functions (Cont.)

- ▶ Figure 7.28 provides a generic example of the mechanics of defining and using an array of pointers to functions.
- ▶ We define three functions—`function1`, `function2` and `function3`—that each take an integer argument and return nothing.
- ▶ We store pointers to these three functions in array `f`, which is defined in line 14.



```
1  /* Fig. 7.28: fig07_28.c
2     Demonstrating an array of pointers to functions */
3  #include <stdio.h>
4
5  /* prototypes */
6  void function1( int a );
7  void function2( int b );
8  void function3( int c );
9
10 int main( void )
11 {
12     /* initialize array of 3 pointers to functions that each take an
13        int argument and return void */
14     void (*f[ 3 ])( int ) = { function1, function2, function3 };
15
16     int choice; /* variable to hold user's choice */
17
18     printf( "Enter a number between 0 and 2, 3 to end: " );
19     scanf( "%d", &choice );
20
```

Fig. 7.28 | Demonstrating an array of pointers to functions. (Part I of 3.)



```
21  /* process user's choice */
22  while ( choice >= 0 && choice < 3 ) {
23
24      /* invoke function at location choice in array f and pass
25       choice as an argument */
26      (*f[ choice ])( choice );
27
28      printf( "Enter a number between 0 and 2, 3 to end: " );
29      scanf( "%d", &choice );
30  } /* end while */
31
32  printf( "Program execution completed.\n" );
33  return 0; /* indicates successful termination */
34 } /* end main */
35
36 void function1( int a )
37 {
38     printf( "You entered %d so function1 was called\n\n", a );
39 } /* end function1 */
40
41 void function2( int b )
42 {
43     printf( "You entered %d so function2 was called\n\n", b );
44 } /* end function2 */
```

Fig. 7.28 | Demonstrating an array of pointers to functions. (Part 2 of 3.)



```
45
46 void function3( int c )
47 {
48     printf( "You entered %d so function3 was called\n\n", c );
49 } /* end function3 */
```

```
Enter a number between 0 and 2, 3 to end: 0
You entered 0 so function1 was called
```

```
Enter a number between 0 and 2, 3 to end: 1
You entered 1 so function2 was called
```

```
Enter a number between 0 and 2, 3 to end: 2
You entered 2 so function3 was called
```

```
Enter a number between 0 and 2, 3 to end: 3
Program execution completed.
```

Fig. 7.28 | Demonstrating an array of pointers to functions. (Part 3 of 3.)

7.12 Pointers to Functions (Cont.)

- ▶ The definition is read beginning in the leftmost set of parentheses, “`f` is an array of 3 pointers to functions that each take an `int` as an argument and return `void`.” The array is initialized with the names of the three functions.
- ▶ When the user enters a value between 0 and 2, the value is used as the subscript into the array of pointers to functions.
- ▶ In the function call (line 26), `f[choice]` selects the pointer at location `choice` in the array.
- ▶ The pointer is dereferenced to call the function, and `choice` is passed as the argument to the function.
- ▶ Each function prints its argument’s value and its function name to demonstrate that the function is called correctly.