



Chapter 7

Pointers

C++ How to Program,
Late Objects Version, 7/e



OBJECTIVES

In this chapter you'll learn:

- What pointers are.
- The similarities and differences between pointers and references, and when to use each.
- To use pointers to pass arguments to functions by reference.
- The close relationships between pointers and arrays.
- To use arrays of pointers.
- Basic pointer-based string processing.
- To use pointers to functions.



-
- 7.1** Introduction
 - 7.2** Pointer Variable Declarations and Initialization
 - 7.3** Pointer Operators
 - 7.4** Pass-by-Reference with Pointers
 - 7.5** Using `const` with Pointers
 - 7.6** Selection Sort Using Pass-by-Reference
 - 7.7** `sizeof` Operator
 - 7.8** Pointer Expressions and Pointer Arithmetic
 - 7.9** Relationship Between Pointers and Arrays
 - 7.10** Pointer-Based String Processing
 - 7.11** Arrays of Pointers
 - 7.12** Function Pointers
 - 7.13** Wrap-Up
-



7.1 Introduction

- ▶ Pointers also enable pass-by-reference and can be used to create and manipulate dynamic data structures that can grow and shrink, such as linked lists, queues, stacks and trees.
- ▶ This chapter explains basic pointer concepts and reinforces the intimate relationship among arrays and pointers.



7.2 Pointer Variable Declarations and Initialization

- ▶ A pointer contains the memory address of a variable that, in turn, contains a specific value.
- ▶ In this sense, a variable name **directly references a value**, and a pointer **indirectly references a value**.
- ▶ Referencing a value through a pointer is called **indirection**.
- ▶ Diagrams typically represent a pointer as an arrow from the variable that contains an address to the variable located at that address in memory.

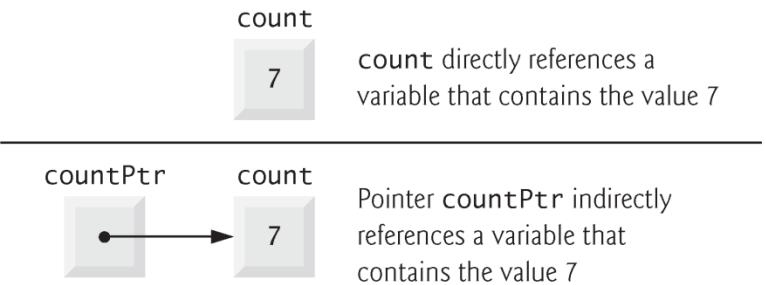


Fig. 7.1 | Directly and indirectly referencing a variable.



7.2 Pointer Variable Declarations and Initialization (cont.)

- ▶ The declaration
 - `int *countPtr, count;` declares the variable `countPtr` to be of type `int *` (i.e., a pointer to an `int` value) and is read (right to left), “`countPtr` is a pointer to `int`.
◦ Variable `count` in the preceding declaration is declared to be an `int`, not a pointer to an `int`.
◦ The `*` in the declaration applies only to `countPtr`.
◦ Each variable being declared as a pointer must be preceded by an asterisk (`*`).
▶ When `*` appears in a declaration, it isn’t an operator; rather, it indicates that the variable being declared is a pointer.
▶ Pointers can be declared to point to objects of any data type.



Common Programming Error 7.1

*Assuming that the * used to declare a pointer distributes to all variable names in a declaration's comma-separated list of variables can lead to errors. Each pointer must be declared with the * prefixed to the name (either with or without spaces in between—the compiler ignores the space). Declaring only one variable per declaration helps avoid these types of errors and improves program readability.*



Good Programming Practice 7.1

Although it isn't a requirement, including the letters `Ptr` in a pointer variable name makes it clear that the variable is a pointer and that it must be handled accordingly.



7.2 Pointer Variable Declarations and Initialization (cont.)

- ▶ Pointers should be initialized either when they're declared or in an assignment.
- ▶ A pointer may be initialized to 0, **NULL** or an address of the corresponding type.
- ▶ A pointer with the value 0 or **NULL** points to nothing and is known as a **null pointer**.
 - **NULL** is equivalent to 0, but in C++, 0 is used by convention.
- ▶ The value 0 is the only integer value that can be assigned directly to a pointer variable without first casting the integer to a pointer type.



Error-Prevention Tip 7.1

Initialize pointers to prevent pointing to unknown or uninitialized areas of memory.



7.3 Pointer Operators

- ▶ The **address operator** (`&`) is a unary operator that obtains the memory address of its operand.
- ▶ Assuming the declarations
 - `int y = 5; // declare variable y`
 - `int *yPtr; // declare pointer variable yPtr`the statement
 - `yPtr = &y; // assign address of y to yPtr`assigns the address of the variable `y` to pointer variable `yPtr`.
- ▶ Figure 7.2 shows a schematic representation of memory after the preceding assignment.

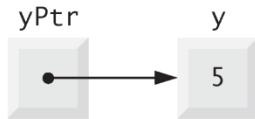


Fig. 7.2 | Graphical representation of a pointer pointing to a variable in memory.



7.3 Pointer Operators (cont.)

- ▶ Figure 7.3 shows another pointer representation in memory with integer variable `y` stored at memory location 600000 and pointer variable `yPtr` stored at memory location 500000.
- ▶ The operand of the address operator must be an *lvalue*; the address operator cannot be applied to constants or to expressions that do not result in references.
- ▶ The `*` operator, commonly referred to as the **indirection operator** or **dereferencing operator**, returns a synonym for the object to which its pointer operand points.
 - Called **dereferencing a pointer**
- ▶ A dereferenced pointer may also be used on the left side of an assignment.



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



Common Programming Error 7.2

Dereferencing an uninitialized pointer could cause a fatal execution-time error, or it could accidentally modify important data and allow the program to run to completion, possibly with incorrect results.



Common Programming Error 7.3

*An attempt to dereference a variable that's not a pointer
is a compilation error.*



Common Programming Error 7.4

Dereferencing a null pointer is often a fatal execution-time error.



Portability Tip 7.1

The format in which a pointer is output is compiler dependent—some use hexadecimal integers, some use decimal integers and some use other formats.



```
1 // Fig. 7.4: fig07_04.cpp
2 // Pointer operators & and *.
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     int a; // a is an integer
9     int *aPtr; // aPtr is an int * which is a pointer to an integer
10
11    a = 7; // assigned 7 to a
12    aPtr = &a; // assign the address of a to aPtr
13
14    cout << "The address of a is " << &a
15        << "\n\nThe value of aPtr is " << aPtr;
16    cout << "\n\nThe value of a is " << a
17        << "\n\nThe value of *aPtr is " << *aPtr;
18    cout << "\n\nShowing that * and & are inverses of "
19        << "each other.\n&*aPtr = " << &*aPtr
20        << "\n*(&aPtr) = " << *(&aPtr) << endl;
21 } // end main
```

Fig. 7.4 | Pointer operators & and *. (Part 1 of 2.)



The address of a is 0012F580
The value of aPtr is 0012F580

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

Showing that * and & are inverses of each other.
 $\&*a\text{Ptr} = 0012F580$
 $*\&a\text{Ptr} = 0012F580$

Fig. 7.4 | Pointer operators & and *. (Part 2 of 2.)



7.3 Pointer Operators (cont.)

- ▶ The & and * operators are inverses of one another.
- ▶ Figure 7.5 lists the precedence and associativity of the operators introduced to this point.
- ▶ The address (&) and dereferencing operator (*) are unary operators on the third level.

Operators	Associativity	Type
<code>() []</code>	left to right	highest
<code>++ -- static_cast< type >(operand)</code>	left to right	unary (postfix)
<code>++ -- + - ! & *</code>	right to left	unary (prefix)
<code>* / %</code>	left to right	multiplicative
<code>+ -</code>	left to right	additive
<code><< >></code>	left to right	insertion/extraction
<code>< <= > >=</code>	left to right	relational
<code>== !=</code>	left to right	equality
<code>&&</code>	left to right	logical AND
<code> </code>	left to right	logical OR
<code>? :</code>	right to left	conditional
<code>= += -= *= /= %=</code>	right to left	assignment
<code>,</code>	left to right	comma

Fig. 7.5 | Operator precedence and associativity.



7.4 Pass-by-Reference with Pointers

- ▶ There are three ways in C++ to pass arguments to a function—**pass-by-value**, **pass-by-reference with reference arguments** and **pass-by-reference with pointer arguments**.
- ▶ In this section, we explain pass-by-reference with pointer arguments.
- ▶ Pointers, like references, can be used to modify one or more variables in the caller or to pass pointers to large data objects to avoid the overhead of passing the objects by value.
- ▶ In C++, you can use pointers and the indirection operator (*) to accomplish pass-by-reference.



7.4 Pass-by-Reference with Pointers (cont.)

- ▶ When calling a function with an argument that should be modified, the address of the argument is passed.
- ▶ When the address of a variable is passed to a function, the indirection operator (*) can be used in the function to form a synonym for the name of the variable (i.e., an *lvalue*)—this in turn can be used to modify the variable’s value at that location in the caller’s memory.
- ▶ Figure 7.6 and Fig. 7.7 present two versions of a function that cubes an integer—**cubeByValue** and **cubeByReference**.



```
1 // Fig. 7.6: fig07_06.cpp
2 // Pass-by-value used to cube a variable's value.
3 #include <iostream>
4 using namespace std;
5
6 int cubeByValue( int ); // prototype
7
8 int main()
9 {
10    int number = 5;
11
12    cout << "The original value of number is " << number;
13
14    number = cubeByValue( number ); // pass number by value to cubeByValue
15    cout << "\nThe new value of number is " << number << endl;
16 } // end main
17
18 // calculate and return cube of integer argument
19 int cubeByValue( int n )
20 {
21    return n * n * n; // cube local variable n and return result
22 } // end function cubeByValue
```

Fig. 7.6 | Pass-by-value used to cube a variable's value. (Part I of 2.)

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

Fig. 7.6 | Pass-by-value used to cube a variable's value. (Part 2 of 2.)



7.4 Pass-by-Reference with Pointers (cont.)

- ▶ Figure 7.7 passes the variable **number** to function **cubeByReference** using pass-by-reference with a pointer argument—the address of **number** is passed to the function.
- ▶ The function dereferences the pointer and cubes the value to which **nPtr** points.
 - This directly changes the value of **number** in **main**.
- ▶ Figures 7.8–7.9 analyze graphically the execution of the programs in Fig. 7.6 and Fig. 7.7, respectively.



Common Programming Error 7.5

Not dereferencing a pointer when it's necessary to do so to obtain the value to which the pointer points is an error.



```
1 // Fig. 7.7: fig07_07.cpp
2 // Pass-by-reference with a pointer argument used to cube a
3 // variable's value.
4 #include <iostream>
5 using namespace std;
6
7 void cubeByReference( int * ); // prototype
8
9 int main()
10 {
11     int number = 5;
12
13     cout << "The original value of number is " << number;
14
15     cubeByReference( &number ); // pass number address to cubeByReference
16
17     cout << "\nThe new value of number is " << number << endl;
18 } // end main
19
```

Fig. 7.7 | Pass-by-reference with a pointer argument used to cube a variable's value.
(Part I of 2.)

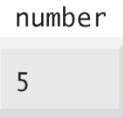
```
20 // calculate cube of *nPtr; modifies variable number in main
21 void cubeByReference( int *nPtr )
22 {
23     *nPtr = *nPtr * *nPtr * *nPtr; // cube *nPtr
24 } // end function cubeByReference
```

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

Fig. 7.7 | Pass-by-reference with a pointer argument used to cube a variable's value.
(Part 2 of 2.)

Step 1: Before main calls cubeByValue:

```
int main()
{
    int number = 5;
    number = cubeByValue( number );
}
```

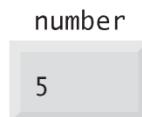


```
int cubeByValue( int n )
{
    return n * n * n;
}
```



Step 2: After cubeByValue receives the call:

```
int main()
{
    int number = 5;
    number = cubeByValue( number );
}
```



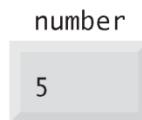
```
int cubeByValue( int n )
{
    return n * n * n;
}
```



Fig. 7.8 | Pass-by-value analysis of the program of Fig. 7.6. (Part I of 3.)

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

```
int main()
{
    int number = 5;
    number = cubeByValue( number );
}
```

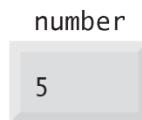


```
int cubeByValue( int n )
{
    return n * n * n;
}
```



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

```
int main()
{
    int number = 5;      125
    number = cubeByValue( number );
}
```



```
int cubeByValue( int n )
{
    return n * n * n;
}
```

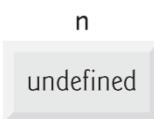


Fig. 7.8 | Pass-by-value analysis of the program of Fig. 7.6. (Part 2 of 3.)

Step 5: After main completes the assignment to number:

```
int main()
{
    int number = 5;
    125
    number = cubeByValue( number );
}
```

number

125

```
int cubeByValue( int n )
{
    return n * n * n;
}
```

n

undefined

Fig. 7.8 | Pass-by-value analysis of the program of Fig. 7.6. (Part 3 of 3.)

Step 1: Before main calls cubeByReference:

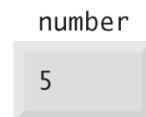
```
int main()
{
    int number = 5;
    cubeByReference( &number );
}
```



```
void cubeByReference( int *nPtr )
{
    *nPtr = *nPtr * *nPtr * *nPtr;
}
nPtr  
undefined
```

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

```
int main()
{
    int number = 5;
    cubeByReference( &number );
}
```



```
void cubeByReference( int *nPtr )
{
    *nPtr = *nPtr * *nPtr * *nPtr;
}
nPtr  
call establishes this pointer
```

Fig. 7.9 | Pass-by-reference analysis (with a pointer argument) of the program of Fig. 7.7. (Part 1 of 2.)



Software Engineering Observation 7.1

Use pass-by-value to pass arguments to a function unless the caller explicitly requires that the called function directly modify the value of the argument variable in the caller. This is another example of the principle of least privilege.



7.4 Pass-by-Reference with Pointers (cont.)

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



7.5 Using `const` with Pointers

- ▶ Many possibilities exist for using (or not using) `const` with function parameters.
- ▶ Principle of least privilege
 - Always give a function enough access to the data in its parameters to accomplish its specified task, but no more.



Software Engineering Observation 7.2

If a value does not (or should not) change in the body of a function to which it's passed, the parameter should be declared const.



Error-Prevention Tip 7.2

Before using a function, check its function prototype to determine the parameters that it can modify.



7.5 Using `const` with Pointers (cont.)

- ▶ There are four ways to pass a pointer to a function
 - a nonconstant pointer to nonconstant data
 - a nonconstant pointer to constant data (Fig. 7.10)
 - a constant pointer to nonconstant data (Fig. 7.11)
 - a constant pointer to constant data (Fig. 7.12)
- ▶ Each combination provides a different level of access privilege.



7.5 Using `const` with Pointers (cont.)

- ▶ The highest access is granted by a **nonconstant pointer to nonconstant data**
 - The data can be modified through the dereferenced pointer, and the pointer can be modified to point to other data.
- ▶ Such a pointer's declaration (e.g., `int *countPtr`) does not include `const`.



7.5 Using `const` with Pointers (cont.)

- ▶ A **nonconstant pointer to constant data**
 - A pointer that can be modified to point to any data item of the appropriate type, but the data to which it points cannot be modified through that pointer.
- ▶ Might be used to receive an array argument to a function that will process each array element, but should not be allowed to modify the data.
- ▶ Any attempt to modify the data in the function results in a compilation error.
- ▶ Sample declaration:
 - `const int *countPtr;`
 - Read from right to left as “`countPtr` is a pointer to an integer constant.”
- ▶ Figure 7.10 demonstrates the compilation error messages produced when attempting to compile a function that receives a nonconstant pointer to constant data, then tries to use that pointer to modify the data.

```
1 // Fig. 7.10: fig07_10.cpp
2 // Attempting to modify data through a
3 // nonconstant pointer to constant data.
4
5 void f( const int * ); // prototype
6
7 int main()
8 {
9     int y;
10
11     f( &y ); // f attempts illegal modification
12 } // end main
13
14 // xPtr cannot modify the value of constant variable to which it points
15 void f( const int *xPtr )
16 {
17     *xPtr = 100; // error: cannot modify a const object
18 } // end function f
```

Fig. 7.10 | Attempting to modify data through a nonconstant pointer to constant data. (Part I of 2.)



Microsoft Visual C++ compiler error message:

```
c:\cpphttp7_examples\ch07\Fig07_10\fig07_10.cpp(17) :  
error C3892: 'xPtr' : you cannot assign to a variable that is const
```

GNU C++ compiler error message:

```
fig07_10.cpp: In function `void f(const int*)':  
fig07_10.cpp:17: error: assignment of read-only location
```

Fig. 7.10 | Attempting to modify data through a nonconstant pointer to constant data. (Part 2 of 2.)



Performance Tip 7.1

If they do not need to be modified by the called function, pass large objects using pointers to constant data or references to constant data, to obtain the performance benefits of pass-by-reference.



Software Engineering Observation 7.3

Pass large objects using pointers to constant data, or references to constant data, to obtain the security of pass-by-value.



7.5 Using `const` with Pointers (cont.)

- ▶ A **constant pointer to nonconstant data** is a pointer that always points to the same memory location; the data at that location can be modified through the pointer.
- ▶ An example of such a pointer is an array name, which 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 nonconstant data can be used to receive an array as an argument to a function that accesses array elements using array subscript notation.
- ▶ Pointers that are declared **const** must be initialized when they're declared.
- ▶ If the pointer is a function parameter, it's initialized with a pointer that's passed to the function.



Common Programming Error 7.6

Not initializing a pointer that's declared `const` is a compilation error.



```
1 // Fig. 7.11: fig07_11.cpp
2 // Attempting to modify a constant pointer to nonconstant data.
3
4 int main()
5 {
6     int x, y;
7
8     // ptr is a constant pointer to an integer that can
9     // be modified through ptr, but ptr always points to the
10    // same memory location.
11    int * const ptr = &x; // const pointer must be initialized
12
13    *ptr = 7; // allowed: *ptr is not const
14    ptr = &y; // error: ptr is const; cannot assign to it a new address
15 } // end main
```

Fig. 7.11 | Attempting to modify a constant pointer to nonconstant data. (Part 1 of 2.)



Microsoft Visual C++ compiler error message:

```
c:\cpphttp7_examples\ch07\Fig07_11\fig07_11.cpp(14) : error C3892: 'ptr' :  
you cannot assign to a variable that is const
```

GNU C++ compiler error message:

```
fig07_11.cpp: In function `int main()':  
fig07_11.cpp:14: error: assignment of read-only variable `ptr'
```

Fig. 7.11 | Attempting to modify a constant pointer to nonconstant data. (Part 2 of 2.)



7.5 Using `const` with Pointers (cont.)

- ▶ The minimum 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 location cannot be modified via the pointer.
 - This is how an array should be passed to a function that *only reads the array, using array subscript notation, and does not modify the array*.
- ▶ The program of Fig. 7.12 declares pointer variable `ptr` to be of type `const int * const` (line 13).
- ▶ This declaration 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 and when an attempt is made to modify the address stored in the pointer variable.



```
1 // Fig. 7.12: fig07_12.cpp
2 // Attempting to modify a constant pointer to constant data.
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     int x = 5, y;
9
10    // ptr is a constant pointer to a constant integer.
11    // ptr always points to the same location; the integer
12    // at that location cannot be modified.
13    const int *const ptr = &x;
14
15    cout << *ptr << endl;
16
17    *ptr = 7; // error: *ptr is const; cannot assign new value
18    ptr = &y; // error: ptr is const; cannot assign new address
19 } // end main
```

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



Microsoft Visual C++ compiler error message:

```
c:\cpphttp7_examples\ch07\Fig07_12\fig07_12.cpp(17) : error C3892: 'ptr' :  
    you cannot assign to a variable that is const  
c:\cpphttp7_examples\ch07\Fig07_12\fig07_12.cpp(18) : error C3892: 'ptr' :  
    you cannot assign to a variable that is const
```

GNU C++ compiler error message:

```
fig07_12.cpp: In function `int main()':  
fig07_12.cpp:17: error: assignment of read-only location  
fig07_12.cpp:18: error: assignment of read-only variable `ptr'
```

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



7.6 Selection Sort Using Pass-by-Reference

► Selection sort

- Easy-to-program, but inefficient, sorting algorithm.
- The first iteration of the algorithm selects the smallest element in the array and swaps it with the first element.
- The second iteration selects the second-smallest element (which is the smallest element of the remaining elements) and swaps it with the second element.
- The algorithm continues until the last iteration selects the second-largest element and swaps it with the second-to-last index, leaving the largest element in the last index.
- After the i^{th} iteration, the smallest i items of the array will be sorted into increasing order in the first i elements of the array.



```
1 // Fig. 7.13: fig07_13.cpp
2 // Selection sort with pass-by-reference. This program puts values into an
3 // array, sorts them into ascending order and prints the resulting array.
4 #include <iostream>
5 #include <iomanip>
6 using namespace std;
7
8 void selectionSort( int * const, const int ); // prototype
9 void swap( int * const, int * const ); // prototype
10
11 int main()
12 {
13     const int arraySize = 10;
14     int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
15
16     cout << "Data items in original order\n";
17
18     for ( int i = 0; i < arraySize; i++ )
19         cout << setw( 4 ) << a[ i ];
20
21     selectionSort( a, arraySize ); // sort the array
22
23     cout << "\nData items in ascending order\n";
24 }
```

Fig. 7.13 | Selection sort with pass-by-reference. (Part I of 3.)



```
25     for ( int j = 0; j < arraySize; j++ )
26         cout << setw( 4 ) << a[ j ];
27
28     cout << endl;
29 } // end main
30
31 // function to sort an array
32 void selectionSort( int * const array, const int size )
33 {
34     int smallest; // index of smallest element
35
36     // loop over size - 1 elements
37     for ( int i = 0; i < size - 1; i++ )
38     {
39         smallest = i; // first index of remaining array
40
41         // loop to find index of smallest element
42         for ( int index = i + 1; index < size; index++ )
43
44             if ( array[ index ] < array[ smallest ] )
45                 smallest = index;
46
47             swap( &array[ i ], &array[ smallest ] );
48     } // end if
49 } // end function selectionSort
```

Fig. 7.13 | Selection sort with pass-by-reference. (Part 2 of 3.)

```
50
51 // swap values at memory locations to which
52 // element1Ptr and element2Ptr point
53 void swap( int * const element1Ptr, int * const element2Ptr )
54 {
55     int hold = *element1Ptr;
56     *element1Ptr = *element2Ptr;
57     *element2Ptr = hold;
58 } // 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. 7.13 | Selection sort with pass-by-reference. (Part 3 of 3.)



7.6 Selection Sort Using Pass-by-Reference (cont.)

- ▶ Remember that C++ enforces information hiding between functions, so function `swap` does not have access to individual array elements in `selectionSort`.
- ▶ Because `selectionSort` wants `swap` to have access to the array elements to be swapped, `selectionSort` passes each of these elements to `swap` by reference.
 - The address of each array element is passed explicitly.
- ▶ When `swap` references `*element1Ptr`, it's actually referencing `array[i]` in `selectionSort`.
- ▶ When `swap` references `*element2Ptr`, it's actually referencing `array[smallest]` in `selectionSort`.



Software Engineering Observation 7.4

When passing an array to a function, also pass the size of the array (rather than building into the function knowledge of the array size)—this makes the function more reusable.



7.7 sizeof Operator

- ▶ The unary operator `sizeof` determines the size of an array (or of any other data type, variable or constant) in bytes during program compilation.
- ▶ When applied to the name of an array, the `sizeof` operator returns the total number of bytes in the array as a value of type `size_t`.
- ▶ When applied to a pointer parameter in a function that receives an array as an argument, the `sizeof` operator returns the size of the pointer in bytes—not the size of the array.



Common Programming Error 7.7

Using the `sizeof` operator in a function to find the size in bytes of an array parameter results in the size in bytes of a pointer, not the size in bytes of the array.



```
1 // Fig. 7.14: fig07_14.cpp
2 // Sizeof operator when used on an array name
3 // returns the number of bytes in the array.
4 #include <iostream>
5 using namespace std;
6
7 size_t getSize( double * ); // prototype
8
9 int main()
10 {
11     double array[ 20 ]; // 20 doubles; occupies 160 bytes on our system
12
13     cout << "The number of bytes in the array is " << sizeof( array );
14
15     cout << "\nThe number of bytes returned by getSize is "
16         << getSize( array ) << endl;
17 } // end main
18
19 // return size of ptr
20 size_t getSize( double *ptr )
21 {
22     return sizeof( ptr );
23 } // end function getSize
```

Fig. 7.14 | `sizeof` operator when applied to an array name returns the number of bytes in the array. (Part I of 2.)



The number of bytes in the array is 160
The number of bytes returned by getSize is 4

Fig. 7.14 | `sizeof` operator when applied 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 using the results of two `sizeof` operations.
- ▶ Consider the following array declaration:
 - `double realArray[22];`
- ▶ To determine the number of elements in the array, the following expression (which is evaluated at compile time) can be used:
 - `sizeof realArray / sizeof(realArray[0])`
- ▶ The expression determines the number of bytes in array `realArray` and divides that value by the number of bytes used in memory to store the array's first element.



7.7 sizeof Operator (cont.)

- ▶ Figure 7.15 uses `sizeof` to calculate the number of bytes used to store most of the standard data types.
- ▶ The output shows that the types `double` and `long double` have the same size.
 - Types may have different sizes based on the platform running the program.



```
1 // Fig. 7.15: fig07_15.cpp
2 // Demonstrating the sizeof operator.
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     char c; // variable of type char
9     short s; // variable of type short
10    int i; // variable of type int
11    long l; // variable of type long
12    float f; // variable of type float
13    double d; // variable of type double
14    long double ld; // variable of type long double
15    int array[ 20 ]; // array of int
16    int *ptr = array; // variable of type int *
17
18    cout << "sizeof c = " << sizeof c
19        << "\nsizeof(char) = " << sizeof( char )
20        << "\nsizeof s = " << sizeof s
21        << "\nsizeof(short) = " << sizeof( short )
22        << "\nsizeof i = " << sizeof i
```

Fig. 7.15 | `sizeof` operator used to determine standard data type sizes. (Part I of 2.)



```
23     << "\tsizeof(int) = " << sizeof( int )
24     << "\nsizeof l = " << sizeof l
25     << "\tsizeof(long) = " << sizeof( long )
26     << "\nsizeof f = " << sizeof f
27     << "\tsizeof(float) = " << sizeof( float )
28     << "\nsizeof d = " << sizeof d
29     << "\tsizeof(double) = " << sizeof( double )
30     << "\nsizeof ld = " << sizeof ld
31     << "\tsizeof(long double) = " << sizeof( long double )
32     << "\nsizeof array = " << sizeof array
33     << "\nsizeof ptr = " << sizeof ptr << endl;
34 } // 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. 7.15 | sizeof operator used to determine standard data type sizes. (Part 2 of 2.)



Portability Tip 7.2

The number of bytes used to store a particular data type may vary among 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.



7.7 **sizeof** Operator (cont.)

- ▶ Operator **sizeof** can be applied to any expression or type name.
- ▶ When **sizeof** is applied to a variable name (which is not an array name) or other expression, the number of bytes used to store the specific type of the expression's value is returned.
- ▶ The parentheses used with **sizeof** are required only if a type name is supplied as its operand.



Common Programming Error 7.8

Omitting the parentheses in a `sizeof` operation when the operand is a type name is a compilation error.



Performance Tip 7.2

Because `sizeof` is a compile-time unary operator, not an execution-time operator, using `sizeof` does not negatively impact execution performance.



Error-Prevention Tip 7.3

To avoid errors associated with omitting the parentheses around the operand of operator `sizeof`, include parentheses around every `sizeof` operand.



7.8 Pointer Expressions and Pointer Arithmetic

- ▶ Pointers are valid operands in arithmetic expressions, assignment expressions and comparison expressions.
- ▶ **pointer arithmetic**—certain arithmetic operations may be performed on pointers:
 - increment (`++`)
 - decremented (`--`)
 - an integer may be added to a pointer (`+` or `+=`)
 - an integer may be subtracted from a pointer (`-` or `-=`)
 - one pointer may be subtracted from another of the same type



Portability Tip 7.3

Most computers today have four-byte or eight-byte integers. Because the results of pointer arithmetic depend on the size of the objects a pointer points to, pointer arithmetic is machine dependent.

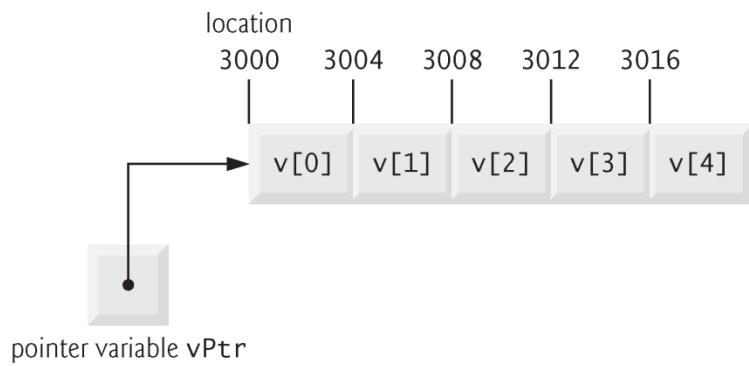


Fig. 7.16 | Array `v` and a pointer variable `int *vPtr` that points to `v`.

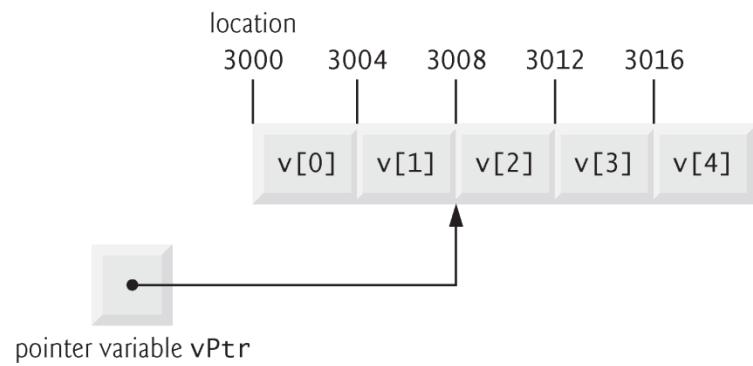


Fig. 7.17 | Pointer `vPtr` after pointer arithmetic.



7.8 Pointer Expressions and Pointer Arithmetic (cont.)

- ▶ Assume that array `int v[5]` has been declared and that its first element is at memory location 3000.
- ▶ Assume that pointer `vPtr` has been initialized to point to `v[0]` (i.e., the value of `vPtr` is 3000).
- ▶ Figure 7.16 diagrams this situation for a machine with four-byte integers.

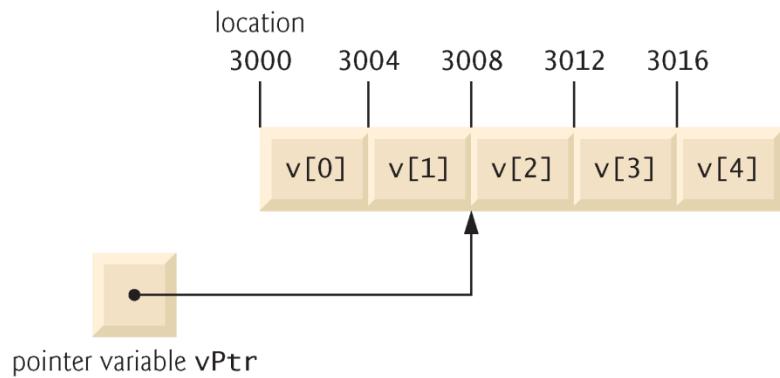


Fig. 8.17 | Pointer `vPtr` after pointer arithmetic.



7.8 Pointer Expressions and Pointer Arithmetic (cont.)

- ▶ In conventional arithmetic, the addition $3000 + 2$ 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 simply incremented or decremented 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.



Common Programming Error 7.9

Using pointer arithmetic on a pointer that does not refer to an array is a logic error.



Common Programming Error 7.10

Subtracting or comparing two pointers that do not refer to elements of the same array is a logic error.



Common Programming Error 7.11

Using pointer arithmetic to move a pointer outside the bounds of an array is a logic error.



7.8 Pointer Expressions and Pointer Arithmetic (cont.)

- ▶ Pointer variables pointing to the same array may be subtracted from one another.
- ▶ For example, if `vPtr` contains the address 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.
- ▶ Pointer arithmetic is meaningless unless performed on a pointer that points to an array.



Software Engineering Observation 7.5

Nonconstant pointer arguments can be passed to constant pointer parameters. This is helpful when the body of a program uses a nonconstant pointer to access data, but does not want that data to be modified by a function called in the body of the program.



7.8 Pointer Expressions and Pointer Arithmetic (cont.)

- ▶ A pointer can be assigned to another pointer if both pointers are of the same type.
- ▶ Otherwise, a cast operator (normally a `reinterpret_cast`; discussed in Section 17.8) must be used to convert the value of the pointer on the right of the assignment to the pointer type on the left of the assignment.
 - Exception to this rule is the pointer to `void` (i.e., `void *`).
- ▶ All pointer types can be assigned to a pointer of type `void *` without casting.



Common Programming Error 7.12

*Assigning a pointer of one type to a pointer of another (other than `void *`) without using a cast (normally a `reinterpret_cast`) is a compilation error.*



Common Programming Error 7.13

*All operations on a void * pointer are compilation errors, except comparing void * pointers with other pointers, casting void * pointers to valid pointer types and assigning addresses to void * pointers.*



7.8 Pointer Expressions and Pointer Arithmetic (cont.)

- ▶ A `void *` pointer cannot be dereferenced.
 - The compiler must know the data type to determine the number of bytes to be dereferenced for a particular pointer—for a pointer to `void`, this number of bytes cannot be determined from the type.



7.8 Pointer Expressions and Pointer Arithmetic (cont.)

- ▶ Pointers can be compared using equality and relational operators.
 - Comparisons using relational operators are meaningless unless the pointers point to members of the same array.
 - Pointer comparisons compare the addresses stored in the pointers.
- ▶ A common use of pointer comparison is determining whether a pointer is 0 (i.e., the pointer is a null pointer—it does not point to anything).



7.9 Relationship Between Pointers and Arrays

- ▶ An array name can be thought of as a constant pointer.
- ▶ Pointers can be used to do any operation involving array subscripting.
- ▶ Assume the following declarations:
 - `int b[5]; // create 5-element int array b`
 - `int *bPtr; // create int pointer bPtr`
- ▶ Because the array name (without a subscript) is a (constant) pointer to the first element of the array, we can set `bPtr` to the address of the first element in array `b` with the statement
 - `bPtr = b; // assign address of array b to bPtr`
- ▶ equivalent to
 - `bPtr = &b[0]; // also assigns address of array b to bPtr`



7.9 Relationship Between Pointers and Arrays (cont.)

- ▶ Array element `b[3]` can alternatively be referenced with the pointer expression
 - `* (bPtr + 3)`
- ▶ The 3 in the preceding expression is the **offset** to the pointer.
- ▶ This notation is referred to as **pointer/offset notation**.
 - The parentheses are necessary, because the precedence of `*` is higher than that of `+`.



7.9 Relationship Between Pointers and Arrays (cont.)

- ▶ 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 name (which is implicitly `const`) can be treated as a pointer and used in pointer arithmetic.
- ▶ 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.



Common Programming Error 7.14

Although array names are pointers to the beginning of the array, array names cannot be modified in arithmetic expressions, because array names are constant pointers.



Good Programming Practice 7.2

For clarity, use array notation instead of pointer notation when manipulating arrays.



7.9 Relationship Between Pointers and Arrays (cont.)

- ▶ Pointers can be subscripted exactly as arrays can.
- ▶ For example, the expression
 - `bPtr[1]`
- ▶ refers to the array element `b[1]`; this expression uses **pointer/subscript notation**.



```
1 // Fig. 7.18: fig07_18.cpp
2 // Using subscripting and pointer notations with arrays.
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     int b[] = { 10, 20, 30, 40 }; // create 4-element array b
9     int *bPtr = b; // set bPtr to point to array b
10
11    // output array b using array subscript notation
12    cout << "Array b printed with:\n\nArray subscript notation\n";
13
14    for ( int i = 0; i < 4; i++ )
15        cout << "b[" << i << "] = " << b[ i ] << '\n';
16
17    // output array b using the array name and pointer/offset notation
18    cout << "\nPointer/offset notation where "
19        << "the pointer is the array name\n";
20
21    for ( int offset1 = 0; offset1 < 4; offset1++ )
22        cout << "*(" << b << offset1 << ") = " << *( b + offset1 ) << '\n';
```

Fig. 7.18 | Referencing array elements with the array name and with pointers. (Part 1 of 3.)



```
23
24 // output array b using bPtr and array subscript notation
25 cout << "\nPointer subscript notation\n";
26
27 for ( int j = 0; j < 4; j++ )
28     cout << "bPtr[" << j << "] = " << bPtr[ j ] << '\n';
29
30 cout << "\nPointer/offset notation\n";
31
32 // output array b using bPtr and pointer/offset notation
33 for ( int offset2 = 0; offset2 < 4; offset2++ )
34     cout << "*(" << bPtr + offset2 << ") = "
35         << *( bPtr + offset2 ) << '\n';
36 } // end main
```

Array b printed with:

Array subscript notation

b[0] = 10
b[1] = 20
b[2] = 30
b[3] = 40

Fig. 7.18 | Referencing array elements with the array name and with pointers. (Part 2 of 3.)



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. 7.18 | Referencing array elements with the array name and with pointers. (Part 3 of 3.)



7.9 Relationship Between Pointers and Arrays (cont.)

- ▶ Figure 7.18 uses the four notations discussed in this section for referring to array elements—array subscript notation, pointer/offset notation with the array name as a pointer, pointer subscript notation and pointer/offset notation with a pointer—to accomplish the same task, namely printing the four elements of the integer array b.



7.10 Pointer-Based String Processing

- ▶ This section introduces C-style, pointer-based strings.
- ▶ C++'s `string` class is preferred for use in new programs, because it eliminates many of the security problems that can be caused by manipulating C strings.
- ▶ We cover C strings here for a deeper understanding of arrays.
- ▶ Also, if you work with legacy C++ programs, you may be required to manipulate these pointer-based strings.



7.10 Pointer-Based String Processing (cont.)

- ▶ Characters are the fundamental building blocks of C++ source programs.
- ▶ Character constant
 - An integer value represented as a character in single quotes.
 - The value of a character constant is the integer value of the character in the machine's character set.
- ▶ A string is a series of characters treated as a single unit.
 - May include letters, digits and various **special characters** such as +, -, *, / and \$.
- ▶ **String literals**, or **string constants**, in C++ are written in double quotation marks
- ▶ A pointer-based string is an array of characters ending with a **null character** ('\0').
- ▶ A string is accessed via a pointer to its first character.



Common Programming Error 7.15

Not allocating sufficient space in a character array to store the null character that terminates a string is an error.



Common Programming Error 7.16

Creating or using a C-style string that does not contain a terminating null character can lead to logic errors.



Error-Prevention Tip 7.4

When storing a string of characters in a character array, be sure that the array is large enough to hold the largest string that will be stored. C++ allows strings of any length to be stored. If a string is longer than the character array in which it's to be stored, characters beyond the end of the array will overwrite data in memory following the array, leading to logic errors.



7.10 Pointer-Based String Processing (cont.)

- ▶ The value of a string literal is the address of its first character, but the **sizeof** a string literal is the length of the string *including the terminating null character*.
- ▶ A string literal may be used as an initializer in the declaration of either a character array or a variable of type **char ***.
- ▶ String literals have **static** storage class (they exist for the duration of the program) and may or may not be shared if the same string literal is referenced from multiple locations in a program.
- ▶ The effect modifying a string literal is **undefined**; thus, you should always declare a pointer to a string literal as **const char ***.
- ▶ When declaring a character array to contain a string, the array must be large enough to store the string and its terminating null character.



Common Programming Error 7.17

Not providing `cin >>` with a character array large enough to store a string typed at the keyboard can result in loss of data in a program and other serious runtime errors.



7.10 Pointer-Based String Processing (cont.)

- ▶ Because a string is an array of characters, we can access individual characters in a string directly with array subscript notation.
- ▶ A string can be read into a character array using stream extraction with `cin`.
- ▶ The `setw` stream manipulator can be used to ensure that the string read into `word` does not exceed the size of the array.
 - Applies only to the next value being input.



7.10 Pointer-Based String Processing (cont.)

- ▶ In some cases, it's desirable to input an entire line of text into a character array.
- ▶ For this purpose, the `cin` object provides the member function `getline`.
- ▶ Three arguments—a character array in which the line of text will be stored, a length and a delimiter character.
- ▶ The function stops reading characters when the delimiter character '`\n`' is encountered, when the end-of-file indicator is entered or when the number of characters read so far is one less than the length specified in the second argument.
- ▶ The third argument to `cin.getline` has '`\n`' as a default value.



Common Programming Error 7.18

*Processing a single character as a `char *` string can lead to a fatal runtime error. A `char *` string is a pointer—probably a respectably large integer. However, a character is a small integer (ASCII values range from 0 to 255). On many systems, dereferencing a `char` value causes an error, because low memory addresses are reserved for special purposes such as operating system interrupt handlers—so “memory access violations” occur.*



Common Programming Error 7.19

Passing a string as an argument to a function when a character is expected is a compilation error.



7.10 Pointer-Based String Processing (cont.)

- ▶ A character array representing a null-terminated string can be output with **cout** and **<<**.
- ▶ The characters of the string are output until a terminating null character is encountered; the null character is not printed.
- ▶ **cin** and **cout** assume that character arrays should be processed as strings terminated by null characters; **cin** and **cout** do not provide similar input and output processing capabilities for other array types.



7.11 Arrays of Pointers

- ▶ Arrays may contain pointers.
- ▶ A common use of such a data structure is to form an array of pointer-based 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 simply a pointer to the first character of a string.
- ▶

```
const char * const suit[ 4 ] =  
    { "Hearts", "Diamonds",  
      "Clubs", "Spades" };
```

 - An array of four elements.
 - Each element is of type “pointer to **char** constant data.”

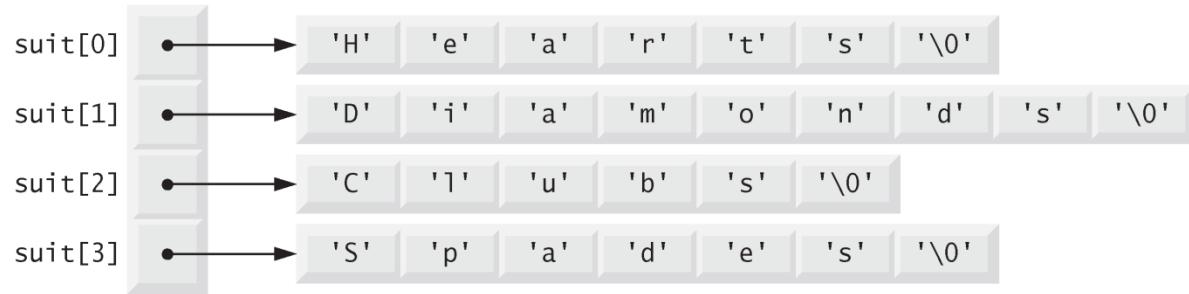


Fig. 7.19 | Graphical representation of the suit array.



7.11 Arrays of Pointers (cont.)

- ▶ String arrays are commonly used with **command-line arguments** that are passed to function **main** when a program begins execution.
- ▶ Such arguments follow the program name when a program is executed from the command line.
- ▶ A typical use of command-line arguments is to pass options to a program.



7.12 Function Pointers

- ▶ A **pointer to a function** contains the function's address in memory.
- ▶ A function's name is actually 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
 - Assigned to other function pointers
 - Used to call the underlying function



7.12 Function Pointers (cont.)

- ▶ To illustrate the use of pointers to functions, Fig. 7.20 modifies the selection sort program of Fig. 7.13.
- ▶ Function `selectionSort` receives a pointer to a function—either function `ascending` or function `descending`—as an argument in addition to the integer array to sort and the size of the array.
- ▶ Functions `ascending` and `descending` determine the sorting order.



```
1 // Fig. 7.20: fig07_20.cpp
2 // Multipurpose sorting program using function pointers.
3 #include <iostream>
4 #include <iomanip>
5 using namespace std;
6
7 // prototypes
8 void selectionSort( int [], const int, bool (*)( int, int ) );
9 void swap( int * const, int * const );
10 bool ascending( int, int ); // implements ascending order
11 bool descending( int, int ); // implements descending order
12
13 int main()
14 {
15     const int arraySize = 10;
16     int order; // 1 = ascending, 2 = descending
17     int counter; // array index
18     int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
19
20     cout << "Enter 1 to sort in ascending order,\n"
21         << "Enter 2 to sort in descending order: ";
22     cin >> order;
23     cout << "\nData items in original order\n";
24 }
```

Fig. 7.20 | Multipurpose sorting program using function pointers. (Part I of 5.)



```
25 // output original array
26 for ( counter = 0; counter < arraySize; counter++ )
27     cout << setw( 4 ) << a[ counter ];
28
29 // sort array in ascending order; pass function ascending
30 // as an argument to specify ascending sorting order
31 if ( order == 1 )
32 {
33     selectionSort( a, arraySize, ascending );
34     cout << "\nData items in ascending order\n";
35 } // end if
36
37 // sort array in descending order; pass function descending
38 // as an argument to specify descending sorting order
39 else
40 {
41     selectionSort( a, arraySize, descending );
42     cout << "\nData items in descending order\n";
43 } // end else part of if...else
44
45 // output sorted array
46 for ( counter = 0; counter < arraySize; counter++ )
47     cout << setw( 4 ) << a[ counter ];
48
```

Fig. 7.20 | Multipurpose sorting program using function pointers. (Part 2 of 5.)



```
49     cout << endl;
50 } // end main
51
52 // multipurpose selection sort; the parameter compare is a pointer to
53 // the comparison function that determines the sorting order
54 void selectionSort( int work[], const int size,
55                     bool (*compare)( int, int ) )
56 {
57     int smallestOrLargest; // index of smallest (or largest) element
58
59     // loop over size - 1 elements
60     for ( int i = 0; i < size - 1; i++ )
61     {
62         smallestOrLargest = i; // first index of remaining vector
63
64         // loop to find index of smallest (or largest) element
65         for ( int index = i + 1; index < size; index++ )
66             if ( !(*compare)( work[ smallestOrLargest ], work[ index ] ) )
67                 smallestOrLargest = index;
68
69         swap( &work[ smallestOrLargest ], &work[ i ] );
70     } // end if
71 } // end function selectionSort
```

Fig. 7.20 | Multipurpose sorting program using function pointers. (Part 3 of 5.)



```
72
73 // swap values at memory locations to which
74 // element1Ptr and element2Ptr point
75 void swap( int * const element1Ptr, int * const element2Ptr )
76 {
77     int hold = *element1Ptr;
78     *element1Ptr = *element2Ptr;
79     *element2Ptr = hold;
80 } // end function swap
81
82 // determine whether element a is less than
83 // element b for an ascending order sort
84 bool ascending( int a, int b )
85 {
86     return a < b; // returns true if a is less than b
87 } // end function ascending
88
89 // determine whether element a is greater than
90 // element b for a descending order sort
91 bool descending( int a, int b )
92 {
93     return a > b; // returns true if a is greater than b
94 } // end function descending
```

Fig. 7.20 | Multipurpose sorting program using function pointers. (Part 4 of 5.)



```
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. 7.20 | Multipurpose sorting program using function pointers. (Part 5 of 5.)



7.12 Function Pointers (cont.)

- ▶ **bool (*compare)(int, int)**
- ▶ This parameter specifies a pointer to a function.
 - The keyword **bool** indicates that the function being pointed to returns a **bool** value.
 - The text **(*compare)** indicates the name of the pointer to the function (the ***** indicates that parameter **compare** is a pointer).
 - The text **(int, int)** indicates that the function pointed to by **compare** takes two integer arguments.
 - Parentheses are needed around ***compare** to indicate that **compare** is a pointer to a function.
- ▶ The corresponding parameter in the function prototype of **selectionSort** is
 - **bool (*)(int, int)**



7.12 Function Pointers (cont.)

- ▶ The function passed to `selectionSort` is called as follows:
 - `(*compare)(work[smallestOrLargest], work[index])`
- ▶ Just as a pointer to a variable is dereferenced to access the value of the variable, a pointer to a function is dereferenced to execute the function.
- ▶ The parentheses around `*compare` are necessary; otherwise, the `*` operator would attempt to dereference the value returned from the function call.
- ▶ Call could have been made without dereferencing the pointer, as in
 - `compare(work[smallestOrLargest], work[index])`
- ▶ which uses the pointer directly as the function name.