5

Pointers and Strings

Objectives

- To be able to use pointers.
- To be able to use pointers to pass arguments to functions by reference.
- To understand the close relationships among pointers, arrays and strings.
- To understand the use of pointers to functions.
- To be able to declare and use arrays of strings.

Addresses are given to us to conceal our whereabouts.
Saki (H. H. Munro)

By indirections find directions out.
William Shakespeare

Many things, having full reference
To one consent, may work contrariously.
William Shakespeare

You will find it a very good practice always to verify your references, sir!
Dr. Routh

You can't trust code that you did not totally create yourself. (Especially code from companies that employ people like me.)
Ken Thompson

Outline

5.1 Introduction

This chapter discusses one of the most powerful features of the C++ programming language, the pointer. Pointers are among C++'s most difficult capabilities to master. In Chapter 3, we saw that references can be used to perform pass-by-reference. Pointers enable programs to simulate pass-by-reference and to create and manipulate dynamic data structures (i.e., data structures that can grow and shrink), such as linked lists, queues, stacks and trees. This chapter explains basic pointer concepts. This chapter also reinforces the intimate relationship among arrays, pointers and strings and includes a substantial collection of string-processing exercises.

Chapter 6 examines the use of pointers with structures and classes. In Chapter 9 and Chapter 10, we will see that the so-called "polymorphic processing" of object-oriented programming is performed with pointers and references. Chapter 17 presents examples of creating and using dynamic data structures.

The view of arrays and strings as pointers derives from C. Later in the book, we will discuss arrays and strings as full-fledged objects.

5.2 Pointer Variable Declarations and Initialization

Pointer variables contain memory addresses as their values. Normally, a variable directly contains a specific value. A pointer, on the other hand, contains the 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. 5.1). Referencing a value through a pointer is often called *indirection.* Note that diagrams typically represent a pointer as an arrow from the variable that contains an address to the variable located at that address in memory.

Pointers, like any other variables, must be declared before they can be used. For example, 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, "**countPtr** is a pointer to **int**" or "**countPtr** points to an object of type **int**." Also, 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 (*). For example, the declaration

```
double *xPtr, *yPtr;
```

indicates that both **xPtr** and **yPtr** are pointers to **double** values. When * appears in a declaration, it is not 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 5.1

*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.*

Good Programming Practice 5.1

*Although it is not a requirement, including the letters **Ptr** in pointer variable names makes it clear that these variables are pointers and that they must be handled appropriately.*

Pointers should be initialized either when they are declared or in an assignment statement. A pointer may be initialized to **0**, ***NULL*** or an address. A pointer with the value **0** or **NULL** points to nothing. Symbolic constant **NULL** is defined in header file **<iostream>** (and in several other standard library header files) to represent the value **0**. Initializing a

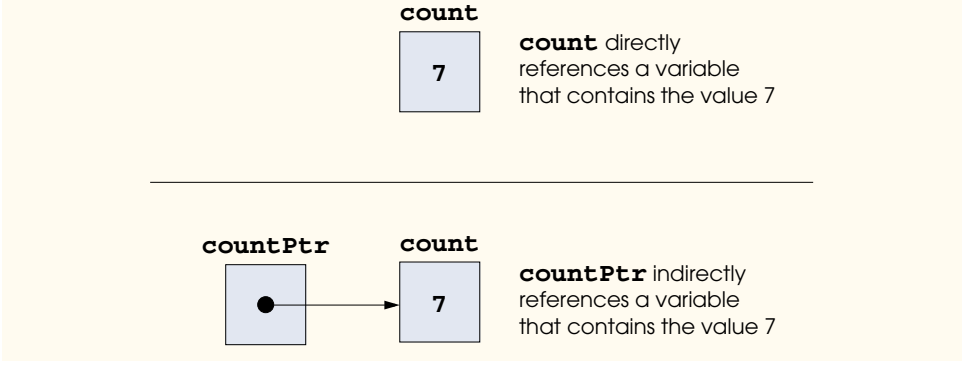

Fig. 5.1 Directly and indirectly referencing a variable.

pointer to **NULL** is equivalent to initializing a pointer to **0**, but in C++, **0** is used by convention. When **0** is assigned, it is 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 without casting the integer to a pointer type first. Assigning a variable's address to a pointer is discussed in Section 5.3.

Testing and Debugging Tip 5.1

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

5.3 Pointer Operators

The *address operator* (**&**) is a unary operator that returns the memory address of its operand. For example, assuming the declarations

```
int y = 5;
int *yPtr;
```

the statement

```
yPtr = &y;
```

assigns the address of the variable **y** to pointer variable **yPtr**. Then variable **yPtr** is said to "point to" **y**. Now, **yPtr** indirectly references variable **y**'s value. Note that the **&** in the preceding assignment statement is not the same as the **&** in a reference variable declaration, which is always preceded by a data-type name.

Figure 5.2 shows a schematic representation of memory after the preceding assignment. In the figure, we show the "pointing relationship" by drawing an arrow from the box that represents the pointer **yPtr** in memory to the box that represents the variable **y** in memory.

Figure 5.3 shows another representation of the pointer in memory, assuming that integer variable **y** is stored at location **600000** and that pointer variable **yPtr** is stored at location **500000**. The operand of the address operator must be an *lvalue* (i.e., something to which a value can be assigned, such as a variable name); the address operator cannot be applied to constants or to expressions that do not result in references.

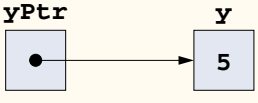

Fig. 5.2 Graphical representation of a pointer pointing to a variable in memory.

Fig. 5.3 Representation of **y** and **yPtr** in memory.

The * *operator*, commonly referred to as the *indirection operator* or *dereferencing operator,* returns a synonym (i.e., an alias or a nickname) for the object to which its pointer operand points. For example (referring again to Fig. 5.2), the statement

```
cout << *yPtr << endl;
```

prints the value of variable **y**, namely, **5**, just as the statement

```
cout << y << endl;
```

would. Using * in this manner is called *dereferencing a pointer.* Note that a dereferenced pointer may also be used on the left side of an assignment statement, as in

```
*yPtr = 9;
```

which would assign **9** to **y** in Fig. 5.3. The dereferenced pointer may also be used to receive an input value as in

```
cin >> *yPtr;
```

The dereferenced pointer is an *lvalue.*

Common Programming Error 5.2

Dereferencing a pointer that has not been properly initialized or that has not been assigned to point to a specific location in memory 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 5.3

An attempt to dereference a variable that is not a pointer is a syntax error.

Common Programming Error 5.4

Dereferencing a ***0*** *pointer is normally a fatal execution-time error.*

The program in Fig. 5.4 demonstrates the & and * pointer operators. Memory locations are output by << in this example as hexadecimal integers. (See Appendix C, Number Systems, for more information on hexadecimal integers.) Note that the hexadecimal memory addresses output by this program are compiler and operating-system dependent.

Portability Tip 5.1

The format in which a pointer is output is machine dependent. Some systems output pointer values as hexadecimal integers, while others use decimal integers.

```
1  // Fig. 5.4: fig05_04.cpp
2  // Using the & and * operators.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
```

Fig. 5.4 Pointer operators & and *. (Part 1 of 2.)

```
 8  int main()
 9  {
10     int a;       // a is an integer
11     int *aPtr;   // aPtr is a pointer to an integer
12
13     a = 7;
14     aPtr = &a;   // aPtr assigned address of a
15
16     cout << "The address of a is " << &a
17          << "\nThe value of aPtr is " << aPtr;
18
19     cout << "\n\nThe value of a is " << a
20          << "\nThe value of *aPtr is " << *aPtr;
21
22     cout << "\n\nShowing that * and & are inverses of "
23          << "each other.\n&*aPtr = " << &*aPtr
24          << "\n*&aPtr = " << *&aPtr << endl;
25
26     return 0;  // indicates successful termination
27
28  } // end main
```

```
The address of a is 0012FED4
The value of aPtr is 0012FED4

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

Showing that * and & are inverses of each other.
&*aPtr = 0012FED4
*&aPtr = 0012FED4
```

Fig. 5.4 Pointer operators **&** and *****. (Part 2 of 2.)

Notice that the address of **a** and the value of **aPtr** are identical in the output, confirming that the address of **a** is indeed assigned to the pointer variable **aPtr**. The **&** and ***** operators are inverses of one another—when they are both applied consecutively to **aPtr** in either order, the same result is printed.

Figure 5.5 lists the precedence and associativity of the operators introduced to this point. Note that the address operator (**&**) and the dereferencing operator (*****) are unary operators on the third level of precedence in the chart.

Operators	Associativity	Type
`()` `[]`	left to right	highest
`++` `--` `static_cast<` *type* `>(` *operand* `)`	left to right	unary
`++` `--` `+` `-` `!` `&` `*`	right to left	unary

Fig. 5.5 Operator precedence and associativity. (Part 1 of 2.)

Operators	Associativity	Type
* / %	left to right	multiplicative
+ -	left to right	additive
<< >>	left to right	insertion/extraction
< <= > >=	left to right	relational
== !=	left to right	equality
&&	left to right	logical AND
||	left to right	logical OR
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment
,	left to right	comma

Fig. 5.5 Operator precedence and associativity. (Part 2 of 2.)

5.4 Calling Functions by Reference

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*. Chapter 3 compared and contrasted pass-by-value and pass-by-reference with reference arguments. This chapter concentrates on pass-by-reference with pointer arguments.

As we saw in Chapter 3, **`return`** can be used to return one value from a called function to a caller (or to return control from a called function without passing back a value). We also saw that arguments can be passed to a function using reference arguments. Such arguments enable the function to modify the original values of the arguments (thus, more than one value can be "returned" from a function). Reference arguments also enable programs to pass large data objects to a function and avoid the overhead of passing the objects by value (which, of course, requires making a copy of the object). Pointers, like references, also 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++, programmers can use pointers and the indirection operator to simulate pass-by-reference (exactly as pass-by-reference is accomplished in C programs, because C does not have references). 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 name of the variable whose value will be modified.

As we saw in Chapter 4, arrays are not passed using operator **`&`**, because the name of the array is the starting location in memory of the array (i.e., an array name is already a pointer). The name of an array is equivalent to **`&arrayName[ 0 ]`**. When the address of a variable is passed to a function, the indirection operator (**`*`**) can be used in the function to form a synonym (i.e., an alias or a nickname) for the name of the variable—this in turn can be used to modify the value of the variable at that location in the caller's memory.

Figure 5.6 and Fig. 5.7 present two versions of a function that cubes an integer—**`cubeByValue`** and **`cubeByReference`**. Figure 5.6 passes variable **`number`** by value to function **`cubeByValue`** (line 17). Function **`cubeByValue`** (lines 26–30) cubes its

argument and passes the new value back to **main** using a **return** statement (line 28). The new value is assigned to **number** in **main**. Note that you have the opportunity to examine the result of the function call before modifying variable **number**'s value. For example, in this program, we could have stored the result of **cubeByValue** in another variable, examined its value and assigned the result to **number** after determining whether the returned value was reasonable.

Figure 5.7 passes the variable **number** to function **cubeByReference** using pass-by-reference with a pointer argument (line 18)—the address of **number** is passed to the function. Function **cubeByReference** (lines 27–31) specifies parameter **nPtr** (a pointer to **int**) to receive its argument. The function dereferences the pointer and cubes the value to which **nPtr** points (line 29). This changes the value of **number** in **main**.

Common Programming Error 5.5

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

```
1  // Fig. 5.6: fig05_06.cpp
2  // Cube a variable using pass-by-value.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  int cubeByValue( int );   // prototype
9
10 int main()
11 {
12    int number = 5;
13
14    cout << "The original value of number is " << number;
15
16    // pass number by value to cubeByValue
17    number = cubeByValue( number );
18
19    cout << "\nThe new value of number is " << number << endl;
20
21    return 0;  // indicates successful termination
22
23 } // end main
24
25 // calculate and return cube of integer argument
26 int cubeByValue( int n )
27 {
28    return n * n * n; // cube local variable n and return result
29
30 } // end function cubeByValue
```

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

Fig. 5.6 Pass-by-value used to cube a variable's value.

```
1  // Fig. 5.7: fig05_07.cpp
2  // Cube a variable using pass-by-reference
3  // with a pointer argument.
4  #include <iostream>
5
6  using std::cout;
7  using std::endl;
8
9  void cubeByReference( int * );   // prototype
10
11 int main()
12 {
13    int number = 5;
14
15    cout << "The original value of number is " << number;
16
17    // pass address of number to cubeByReference
18    cubeByReference( &number );
19
20    cout << "\nThe new value of number is " << number << endl;
21
22    return 0;  // indicates successful termination
23
24 } // end main
25
26 // calculate cube of *nPtr; modifies variable number in main
27 void cubeByReference( int *nPtr )
28 {
29    *nPtr = *nPtr * *nPtr * *nPtr;  // cube *nPtr
30
31 } // end function cubeByReference
```

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

Fig. 5.7 Pass-by-reference with a pointer argument used to cube a variable's value.

A function receiving an address as an argument must define a pointer parameter to receive the address. For example, the header for function **`cubeByReference`** (line 27) specifies that **`cubeByReference`** receives the address of an **`int`** variable (i.e., a pointer to an **`int`**) as an argument, stores the address locally in **`nPtr`** and does not return a value.

The function prototype for **`cubeByReference`** (line 9) contains **`int *`** in parentheses. As with other variable types, it is not necessary to include names of pointer parameters in function prototypes. Parameter names included for documentation purposes are ignored by the compiler.

Figure 5.8 and Fig. 5.9 analyze graphically the execution of the programs in Fig. 5.6 and Fig. 5.7, respectively.

Software Engineering Observation 5.1

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

Before **main** calls **cubeByValue**:

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

   number = cubeByValue( number );
}
```

number: 5

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

n: undefined

After **cubeByValue** receives the call:

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

   number = cubeByValue( number );
}
```

number: 5

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

n: 5

After **cubeByValue** cubes parameter **n** and before **cubeByValue** returns to **main**:

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

   number = cubeByValue( number );
}
```

number: 5

```
int cubeByValue( int n )
{
   return n * n * n;   // 125
}
```

n: 5

After **cubeByValue** returns to **main** and before assigning the result to **number**:

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

   number = cubeByValue( number );   // 125
}
```

number: 5

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

n: undefined

After **main** completes the assignment to **number**:

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

   number = cubeByValue( number );   // 125 ... 125
}
```

number: 125

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

n: undefined

Fig. 5.8 Pass-by-value analysis of the program of Fig. 5.6.

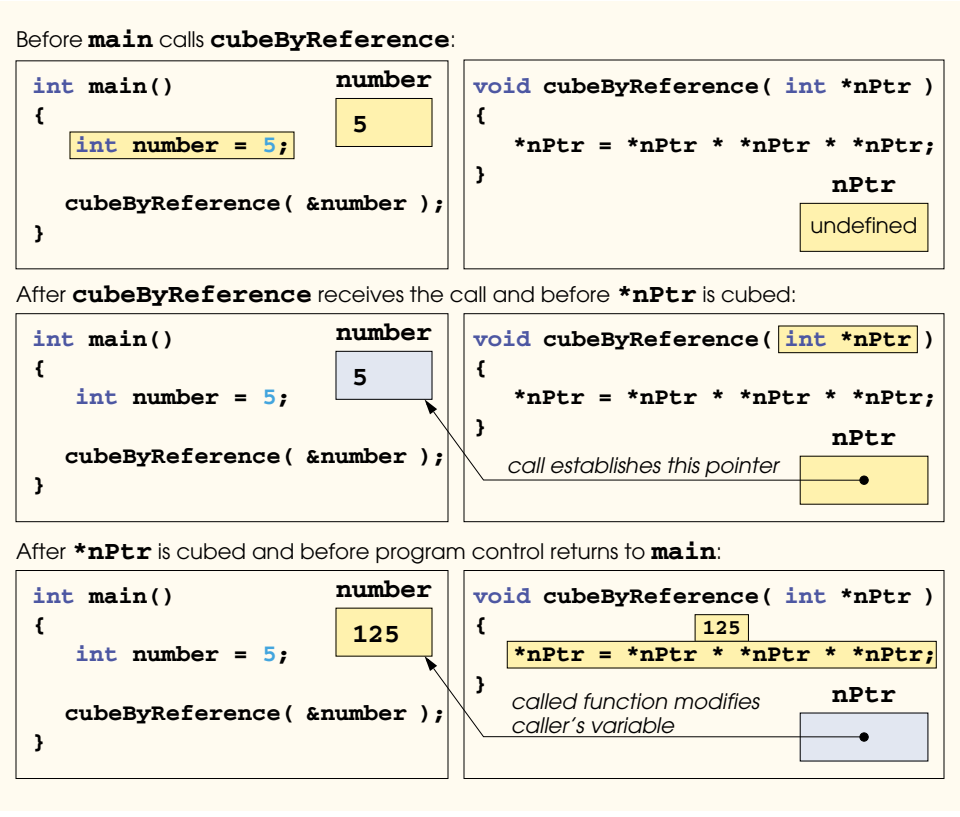

Fig. 5.9 Pass-by-reference analysis (with a pointer argument) of the program of Fig. 5.7.

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 **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 is receiving an array or simply a single variable for which it is to perform pass-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 * const b** (pronounced "**b** is a constant pointer to an integer"—**const** pointers are explained in Section 5.5). Both forms of declaring a function parameter as a single-subscripted array are interchangeable.

5.5 Using `const` with Pointers

The ***const*** *qualifier* enables the programmer to inform the compiler that the value of a particular variable should not be modified.

Software Engineering Observation 5.2

The **const** *qualifier can be used to enforce the principle of least privilege. Using the principle of least privilege to properly design software can greatly reduce debugging time and improper side effects and can make a program easier to modify and maintain.*

Portability Tip 5.2

Although **const** *is well defined in ANSI C and C++, some compilers do not enforce it properly. So a good rule is, "know your compiler."*

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 great opportunities for improvement in the software engineering of old (also called "legacy") C code. Also, many programmers currently using ANSI C and C++ do not use **const** in their programs, because they began programming in early versions of C. These programmers are missing many opportunities for good software engineering.

Many possibilities exist for using (or not using) **const** with function parameters. How do you choose the most appropriate of these 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. This section discusses how to combine **const** with pointer declarations to enforce the principle of least privilege.

Chapter 3 explained that when a function is called using pass-by-value, a copy of the argument (or arguments) in the function call is made and passed to the function. If the copy is modified in the function, the original value is maintained in the caller without 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 the called function manipulates only a copy of the original value.

For example, consider a function that takes a single-subscripted array and its size as arguments and subsequently prints the array. 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 highest subscript of the array so the loop can terminate when the printing completes. The size of the array does not change in the function body, so it should be declared **const**. Of course, because the array is only being printed, it, too, should be declared **const**.

Software Engineering Observation 5.3

If a value does not (or should not) change in the body of a function to which it is passed, the parameter should be declared **const** *to ensure that it is not accidentally modified.*

If an attempt is made to modify a **const** value, a warning or an error is issued, depending on the particular compiler.

Software Engineering Observation 5.4

Only one value can be returned to the caller when pass-by-value is used. To modify multiple values in a calling function, several arguments can be passed by reference.

Good Programming Practice 5.2

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

There are four ways to pass a pointer to a function: a nonconstant pointer to nonconstant data (Fig. 5.10), a nonconstant pointer to constant data (Fig. 5.11 and Fig. 5.12), a

constant pointer to non-constant data (Fig. 5.13) and a constant pointer to constant data (Fig. 5.14). Each combination provides a different level of access privileges.

Nonconstant Pointer to Nonconstant Data

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. Declarations for nonconstant pointers to nonconstant data do not include **const**. Such a pointer can be used to receive a string in a function that changes the pointer value to process (and possibly modify) each character in the string. In Fig. 5.10, function **convertToUppercase** (lines 27–38) declares parameter **sPtr** (line 27) to be a nonconstant pointer to nonconstant data. The function processes the string **phrase** one character at a time (lines 29–36). Function ***islower*** (line 31) takes a character argument and returns true if the character is a lowercase letter and false otherwise. Characters in the range **'a'** through **'z'** are converted to their corresponding uppercase letters by function **toupper** (line 32); others remain unchanged. Function **toupper** takes one character as an argument. If the character is a lowercase letter, the corresponding uppercase letter is returned; otherwise, the original character is returned. Function **toupper** and function **islower** are part of the character handling library **<cctype>**. (See Chapter 18, Bits, Characters, Strings and Structures.) After processing one character, line 34 increments **sPtr** by 1. When operator **++** is applied to a pointer that points to an array, the memory address stored in the pointer is modified to point to the next element of the array (in this case, the next character in the string). Adding one to a pointer is one valid operation in *pointer arithmetic*, which is covered in detail in Section 5.7 and Section 5.8.

```
1  // Fig. 5.10: fig05_10.cpp
2  // Converting lowercase letters to uppercase letters
3  // using a non-constant pointer to non-constant data.
4  #include <iostream>
5
6  using std::cout;
7  using std::endl;
8
9  #include <cctype>    // prototypes for islower and toupper
10
11 void convertToUppercase( char * );
12
13 int main()
14 {
15    char phrase[] = "characters and $32.98";
16
17    cout << "The phrase before conversion is: " << phrase;
18    convertToUppercase( phrase );
19    cout << "\nThe phrase after conversion is:  "
20         << phrase << endl;
21
22    return 0;  // indicates successful termination
23
24 } // end main
25
```

Fig. 5.10 Converting a string to uppercase. (Part 1 of 2.)

```
26  // convert string to uppercase letters
27  void convertToUppercase( char *sPtr )
28  {
29     while ( *sPtr != '\0' ) {   // current character is not '\0'
30
31        if ( islower( *sPtr ) )  // if character is lowercase,
32           *sPtr = toupper( *sPtr );  // convert to uppercase
33
34        ++sPtr;  // move sPtr to next character in string
35
36     } // end while
37
38  } // end function convertToUppercase
```

```
The phrase before conversion is: characters and $32.98
The phrase after conversion is:  CHARACTERS AND $32.98
```

Fig. 5.10 Converting a string to uppercase. (Part 2 of 2.)

Nonconstant Pointer to Constant Data

A nonconstant pointer to constant data is 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. Such a pointer might be used to receive an array argument to a function that will process each element of the array, but should not be allowed to modify the data. For example, function **`printCharacters`** (lines 25–30 of Fig. 5.11) declares parameter **`sPtr`** (line 25) to be of type **`const char *`**. The declaration is read from right to left as "**`sPtr`** is a pointer to a character constant." The body of the function uses a **`for`** structure (lines 27–28) 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. 5.11: fig05_11.cpp
2  // Printing a string one character at a time using
3  // a non-constant pointer to constant data.
4  #include <iostream>
5
6  using std::cout;
7  using std::endl;
8
9  void printCharacters( const char * );
10
11 int main()
12 {
13    char phrase[] = "print characters of a string";
14
15    cout << "The string is:\n";
16    printCharacters( phrase );
17    cout << endl;
18
```

Fig. 5.11 Printing a string one character at a time using a nonconstant pointer to constant data. (Part 1 of 2.)

```
19      return 0;  // indicates successful termination
20
21   } // end main
22
23   // sPtr cannot modify the character to which it points,
24   // i.e., sPtr is a "read-only" pointer
25   void printCharacters( const char *sPtr )
26   {
27      for ( ; *sPtr != '\0'; sPtr++ )   // no initialization
28         cout << *sPtr;
29
30   } // end function printCharacters
```

```
The string is:
print characters of a string
```

Fig. 5.11 Printing a string one character at a time using a nonconstant pointer to constant data. (Part 2 of 2.)

Figure 5.12 demonstrates the syntax 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. 5.12: fig05_12.cpp
2   // Attempting to modify data through a
3   // non-constant pointer to constant data.
4
5   void f( const int * );  // prototype
6
7   int main()
8   {
9      int y;
10
11     f( &y );   // f attempts illegal modification
12
13     return 0;  // indicates successful termination
14
15  } // end main
16
17  // xPtr cannot modify the value of the variable
18  // to which it points
19  void f( const int *xPtr )
20  {
21     *xPtr = 100;  // error: cannot modify a const object
22
23  } // end function f
```

```
d:\cpphtp4_examples\ch05\Fig05_12.cpp(21) : error C2166:
   l-value specifies const object
```

Fig. 5.12 Attempting to modify data through a nonconstant pointer to constant data.

As we know, arrays are aggregate data types that store related data items of the same type under one name. Chapter 6 discusses another form of aggregate data type called a *structure* (sometimes called a *record* in other languages). A structure can store 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 passed to the function by reference. However, structures are always passed by value—a copy of the entire structure is passed. This requires the execution-time overhead of making a copy of each data item in the structure and storing it on the function call stack (the place where the local automatic variables used in the function call are stored while the function is executing). When structure data must be passed to a function, we can use a pointer to constant data (or a reference to constant data) to get the performance of pass-by-reference and the protection of pass-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; the structure itself is not copied. On a machine with four-byte addresses, a copy of four bytes of memory is made rather than a copy of a possibly large structure.

Performance Tip 5.1

Pass large objects such as structures using pointers to constant data, or references to constant data, to obtain the performance benefits of pass-by-reference.

Software Engineering Observation 5.5

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

Constant Pointer to Nonconstant Data

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. 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 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 are declared. (If the pointer is a function parameter, it is initialized with a pointer that is passed to the function.) The program of Fig. 5.13 attempts to modify a constant pointer. Line 12 declares pointer **`ptr`** to be of type **`int * const`**. The declaration in the figure is read from right to left as "**`ptr`** is a constant pointer to an integer." The pointer is initialized with the address of integer variable **`x`**. Line 15 attempts to assign the address of **`y`** to **`ptr`**, but the compiler generates an error message. Note that no error occurs when line 14 assigns the value **`7`** to **`*ptr`**—the nonconstant value to which **`ptr`** points can be modified using **`ptr`**.

```
1  // Fig. 5.13: fig05_13.cpp
2  // Attempting to modify a constant pointer to
3  // non-constant data.
4
5  int main()
6  {
7     int x, y;
8
```

Fig. 5.13 Attempting to modify a constant pointer to nonconstant data. (Part 1 of 2.)

```
9      // ptr is a constant pointer to an integer that can
10     // be modified through ptr, but ptr always points to the
11     // 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
17     return 0;  // indicates successful termination
18
19  } // end main
```

```
d:\cpphtp4_examples\ch05\Fig05_13.cpp(15) : error C2166:
   l-value specifies const object
```

Fig. 5.13 Attempting to modify a constant pointer to nonconstant data. (Part 2 of 2.)

Common Programming Error 5.6

Not initializing a pointer that is declared ***const*** *is a syntax error.*

Constant Pointer to Constant Data

The least amount of 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 using 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. 5.14 declares pointer variable **ptr** to be of type **const int * const** (line 15). 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 (line 19) and when an attempt is made to modify the address stored in the pointer variable (line 20). Note that no errors occur when the program attempts to dereference **ptr**, or when the program attempts to output the value to which **ptr** points (line 17), because neither the pointer nor the data it points to is being modified in this statement.

```
1  // Fig. 5.14: fig05_14.cpp
2  // Attempting to modify a constant pointer to constant data.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  int main()
9  {
10    int x = 5, y;
11
12    // ptr is a constant pointer to a constant integer.
13    // ptr always points to the same location; the integer
14    // at that location cannot be modified.
```

Fig. 5.14 Attempting to modify a constant pointer to constant data. (Part 1 of 2.)

```
15      const int *const ptr = &x;
16
17      cout << *ptr << endl;
18
19      *ptr = 7;  // error: *ptr is const; cannot assign new value
20      ptr = &y;  // error: ptr is const; cannot assign new address
21
22      return 0;  // indicates successful termination
23
24   } // end main
```

```
d:\cpphtp4_examples\ch05\Fig05_14.cpp(19) : error C2166:
   l-value specifies const object
d:\cpphtp4_examples\ch05\Fig05_14.cpp(20) : error C2166:
   l-value specifies const object
```

Fig. 5.14 Attempting to modify a constant pointer to constant data. (Part 2 of 2.)

5.6 Bubble Sort Using Pass-by-Reference

Let us modify the bubble sort program of Fig. 4.16 to use two functions—**bubbleSort** and **swap** (Fig. 5.15). Function **bubbleSort** (lines 40–52) performs the sort of the array. Function **bubbleSort** calls function **swap** (line 50) to exchange the array elements **array[k]** and **array[k + 1]**. 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 by reference to **swap**—the address of each array element is passed explicitly. Although entire arrays are passed by reference, individual array elements are scalars and are ordinarily passed by value. Therefore, **bubbleSort** uses the address operator (**&**) on each array element in the **swap** call (line 50 to effect pass-by-reference). Function **swap** (lines 56–62) receives **&array[k]** in pointer variable **element1Ptr**. Information hiding prevents **swap** from "knowing" the name **array[k]**, but **swap** can use ***element1Ptr** as a synonym for **array[k]**. Thus, when **swap** references ***element1Ptr**, it is actually referencing **array[k]** in **bubbleSort**. Similarly, when **swap** references ***element2Ptr**, it is actually referencing **array[k + 1]** in **bubbleSort**.

```
1  // Fig. 5.15: fig05_15.cpp
2  // This program puts values into an array, sorts the values into
3  // ascending order and prints the resulting array.
4  #include <iostream>
5
6  using std::cout;
7  using std::endl;
8
9  #include <iomanip>
10
```

Fig. 5.15 Bubble sort with pass-by-reference. (Part 1 of 3.)

```
11  using std::setw;
12
13  void bubbleSort( int *, const int );    // prototype
14  void swap( int * const, int * const );  // prototype
15
16  int main()
17  {
18     const int arraySize = 10;
19     int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
20
21     cout << "Data items in original order\n";
22
23     for ( int i = 0; i < arraySize; i++ )
24        cout << setw( 4 ) << a[ i ];
25
26     bubbleSort( a, arraySize );  // sort the array
27
28     cout << "\nData items in ascending order\n";
29
30     for ( int j = 0; j < arraySize; j++ )
31        cout << setw( 4 ) << a[ j ];
32
33     cout << endl;
34
35     return 0;  // indicates successful termination
36
37  } // end main
38
39  // sort an array of integers using bubble sort algorithm
40  void bubbleSort( int *array, const int size )
41  {
42     // loop to control passes
43     for ( int pass = 0; pass < size - 1; pass++ )
44
45        // loop to control comparisons during each pass
46        for ( int k = 0; k < size - 1; k++ )
47
48           // swap adjacent elements if they are out of order
49           if ( array[ k ] > array[ k + 1 ] )
50              swap( &array[ k ], &array[ k + 1 ] );
51
52  } // end function bubbleSort
53
54  // swap values at memory locations to which
55  // element1Ptr and element2Ptr point
56  void swap( int * const element1Ptr, int * const element2Ptr )
57  {
58     int hold = *element1Ptr;
59     *element1Ptr = *element2Ptr;
60     *element2Ptr = hold;
61
62  } // end function swap
```

Fig. 5.15 Bubble sort with pass-by-reference. (Part 2 of 3.)

```
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. 5.15 Bubble sort with pass-by-reference. (Part 3 of 3.)

Even though **swap** is not allowed to use the statements

```
hold = array[ k ];
array[ k ] = array[ k + 1 ];
array[ k + 1 ] = hold;
```

precisely the same effect is achieved by

```
int hold = *element1Ptr;
*element1Ptr = *element2Ptr;
*element2Ptr = hold;
```

in the **swap** function of Fig. 5.15.

Several features of function **bubbleSort** should be noted. The function header (line 40) declares **array** as **int *array**, rather than **int array[]**, to indicate that function **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. The array size remains fixed during the execution of **bubbleSort**. Therefore, **size** is declared **const** to ensure that it is not modified. If the size of the array is modified during the sorting process, the sorting algorithm will not run correctly.

Note that function **bubbleSort** receives the size of the array as a parameter, because 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 array size must be passed separately to the function.

By defining function **bubbleSort** so it receives the array size as a parameter, we enable the function to be used by any program that sorts single-subscripted **int** arrays of arbitrary size. The size of the array could have been programmed directly into the function. This would restrict the use of the function to an array of a specific size and reduce the function's reusability. Only programs processing single-subscripted **int** arrays of the specific size "hard coded" into the function could use the function.

Software Engineering Observation 5.6

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 helps make the function more general. General functions are often reusable in many programs.

C++ provides the *unary operator* **sizeof** to determine 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, as in Fig. 5.16 (line 16), the **sizeof** operator returns the total number of bytes in the array as a value of type **size_t** (which is usually **unsigned int**). The computer we used to compile this program stores variables of type **double** in 8 bytes of memory, and **array** is declared to have 20 elements, so **array** uses 160 bytes in memory.

When applied to a pointer parameter (line 28) in a function that receives an array as an argument, the **sizeof** operator returns the size of the pointer in bytes (4), not the size of the array.

Common Programming Error 5.7

*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.*

The number of elements in an array also can be determined using the results of two **sizeof** operations. For example, consider the following array declaration:

```
double realArray[ 22 ];
```

If variables of data type **double** are stored in eight bytes of memory, array **realArray** contains a total of 176 bytes. To determine the number of elements in the array, the following expression can be used:

```
sizeof realArray / sizeof( double )
```

```
1  // Fig. 5.16: fig05_16.cpp
2  // Sizeof operator when used on an array name
3  // returns the number of bytes in the array.
4  #include <iostream>
5
6  using std::cout;
7  using std::endl;
8
9  size_t getSize( double * );  // prototype
10
11 int main()
12 {
13    double array[ 20 ];
14
15    cout << "The number of bytes in the array is "
16         << sizeof( array );
17
18    cout << "\nThe number of bytes returned by getSize is "
19         << getSize( array ) << endl;
20
21    return 0;  // indicates successful termination
22
23 } // end main
24
25 // return size of ptr
26 size_t getSize( double *ptr )
27 {
28    return sizeof( ptr );
29
30 } // end function getSize
```

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

Fig. 5.16 **sizeof** operator when applied to an array name returns the number of bytes in the array.

The expression determines the number of bytes in array **realArray** and divides that value by the number of bytes used in memory to store a **double** value; the result is the number of elements in **realArray**.

The program of Fig. 5.17 uses the **sizeof** operator to calculate the number of bytes used to store each of the standard data types.

Portability Tip 5.3

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. 5.17: fig05_17.cpp
2  // Demonstrating the sizeof operator.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  int main()
9  {
10    char c;
11    short s;
12    int i;
13    long l;
14    float f;
15    double d;
16    long double ld;
17    int array[ 20 ];
18    int *ptr = array;
19
20    cout << "sizeof c = " << sizeof c
21         << "\tsizeof(char) = " << sizeof( char )
22         << "\nsizeof s = " << sizeof s
23         << "\tsizeof(short) = " << sizeof( short )
24         << "\nsizeof i = " << sizeof i
25         << "\tsizeof(int) = " << sizeof( int )
26         << "\nsizeof l = " << sizeof l
27         << "\tsizeof(long) = " << sizeof( long )
28         << "\nsizeof f = " << sizeof f
29         << "\tsizeof(float) = " << sizeof( float )
30         << "\nsizeof d = " << sizeof d
31         << "\tsizeof(double) = " << sizeof( double )
32         << "\nsizeof ld = " << sizeof ld
33         << "\tsizeof(long double) = " << sizeof( long double )
34         << "\nsizeof array = " << sizeof array
35         << "\nsizeof ptr = " << sizeof ptr
36         << endl;
37
38    return 0;  // indicates successful termination
39
40 } // end main
```

Fig. 5.17 **sizeof** operator used to determine standard data type sizes. (Part 1 of 2.)

```
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. 5.17 **`sizeof`** operator used to determine standard data type sizes. (Part 2 of 2.)

Operator **`sizeof`** can be applied to any variable name, type name or constant value. When **`sizeof`** is applied to a variable name (which is not an array name) or a constant value, the number of bytes used to store the specific type of variable or constant is returned. Note that the parentheses used with **`sizeof`** are required only if a type name is supplied as its operand. The parentheses used with **`sizeof`** are not required when **`sizeof`**'s operand is a variable name or constant. Remember that **`sizeof`** is an operator, not a function, and that it has its effect at compile time, not execution time.

Common Programming Error 5.8

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

Performance Tip 5.2

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

Testing and Debugging Tip 5.2

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

5.7 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 with pointer variables. This section describes the operators that can have pointers as operands and how these operators are used with pointers.

Several 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 **`-=`**) or one pointer may be subtracted from another.

Assume that array **`int v[ 5 ]`** has been declared and that its first element is at location **`3000`** in memory. Assume that pointer **`vPtr`** has been initialized to point to **`v[ 0 ]`** (i.e., that the value of **`vPtr`** is **`3000`**). Figure 5.18 diagrams this situation for a machine with four-byte integers. Note that **`vPtr`** can be initialized to point to array **`v`** with either of the following statements:

```
vPtr = v;
vPtr = &v[ 0 ];
```

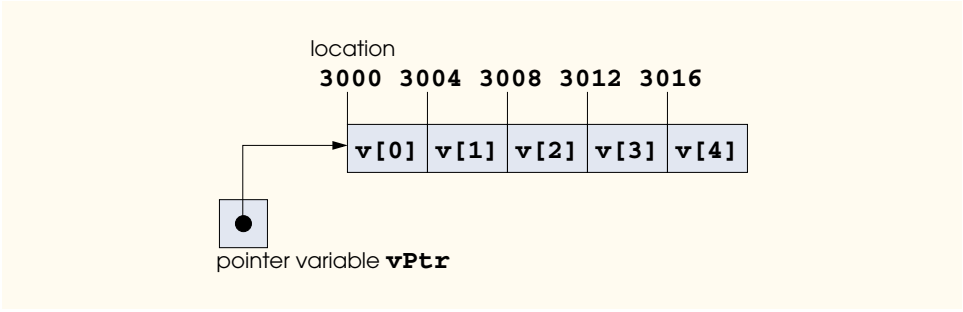

Fig. 5.18 Array `v` and a pointer variable `vPtr` that points to `v`.

Portability Tip 5.4

Most computers today have two-byte or four-byte integers. Some of the newer machines use 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.

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. For example, the statement

```
vPtr += 2;
```

would produce `3008` (`3000 + 2 * 4`), assuming that an `int` is stored in four bytes of memory. In the array `v`, `vPtr` would now point to `v[ 2 ]` (Fig. 5.19). If an integer is stored in two 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 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 one byte long.

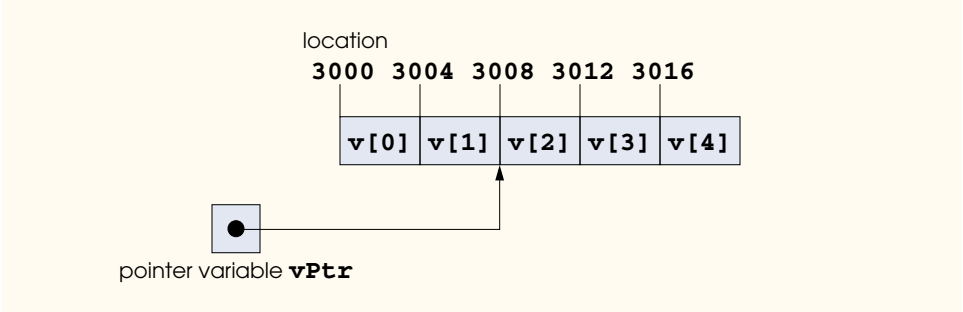

Fig. 5.19 Pointer `vPtr` after pointer arithmetic.

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. Each of the statements

```
++vPtr;
vPtr++;
```

increments the pointer to point to the next element of the array. Each of the statements

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

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

Pointer variables pointing to the same array may be subtracted from one another. 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**. Pointer arithmetic is meaningless unless performed on a pointer that points to an array. We cannot assume that two variables of the same type are stored contiguously in memory unless they are adjacent elements of an array.

Common Programming Error 5.9

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

Common Programming Error 5.10

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

Common Programming Error 5.11

Using pointer arithmetic to increment or decrement a pointer such that the pointer refers to an element past the end of the array or before the beginning of the array is normally a logic error.

A pointer can be assigned to another pointer if both pointers are of the same type. Otherwise, a cast operator 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. The exception to this rule is the pointer to **void** (i.e., **void ***), which is a generic pointer capable of representing any pointer type. All pointer types can be assigned to a pointer of type **void *** without casting. However, a pointer of type **void *** cannot be assigned directly to a pointer of another type—the pointer of type **void *** must first be cast to the proper pointer type.

A **void *** pointer cannot be dereferenced. For example, the compiler "knows" that a pointer to **int** refers to four bytes of memory on a machine with four-byte integers, but a pointer to **void** simply contains a memory address 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. For a pointer to **void**, this number of bytes cannot be determined from the type.

Common Programming Error 5.12

*Assigning a pointer of one type to a pointer of another (other than **void ***) without casting the first pointer to the type of the second pointer is a syntax error.*

Common Programming Error 5.13

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

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 comparison of two pointers pointing to 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 0 (i.e., the pointer does not point to anything).

5.8 Relationship Between Pointers and Arrays

Arrays and pointers are intimately related in C++ and may be used *almost* interchangeably. 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 ];
int *bPtr;
```

Because the array name (without a subscript) is a 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;
```

This is equivalent to taking the address of the first element of the array as follows:

```
bPtr = &b[ 0 ];
```

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. 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.* 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 that **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 name 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. In this case, pointer/offset notation was used with the name of the array as a pointer. Note that the preceding expression 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. For example, the expression

```
bPtr[ 1 ]
```

refers to the array element **b[1]**; this expression uses *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 5.14

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

Good Programming Practice 5.3

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

Figure 5.20 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 print the four elements of the integer array **b**.

```
1  // Fig. 5.20: fig05_20.cpp
2  // Using subscripting and pointer notations with arrays.
3
4  #include <iostream>
5
6  using std::cout;
7  using std::endl;
8
9  int main()
10 {
```

Fig. 5.20 Referencing array elements with the array name and with pointers. (Part 1 of 3.)

```
11      int b[] = { 10, 20, 30, 40 };
12      int *bPtr = b;   // set bPtr to point to array b
13
14      // output array b using array subscript notation
15      cout << "Array b printed with:\n"
16           << "Array subscript notation\n";
17
18      for ( int i = 0; i < 4; i++ )
19         cout << "b[" << i << "] = " << b[ i ] << '\n';
20
21      // output array b using the array name and
22      // pointer/offset notation
23      cout << "\nPointer/offset notation where "
24           << "the pointer is the array name\n";
25
26      for ( int offset1 = 0; offset1 < 4; offset1++ )
27         cout << "*(b + " << offset1 << ") = "
28              << *( b + offset1 ) << '\n';
29
30      // output array b using bPtr and array subscript notation
31      cout << "\nPointer subscript notation\n";
32
33      for ( int j = 0; j < 4; j++ )
34         cout << "bPtr[" << j << "] = " << bPtr[ j ] << '\n';
35
36      cout << "\nPointer/offset notation\n";
37
38      // output array b using bPtr and pointer/offset notation
39      for ( int offset2 = 0; offset2 < 4; offset2++ )
40         cout << "*(bPtr + " << offset2 << ") = "
41              << *( bPtr + offset2 ) << '\n';
42
43      return 0;  // indicates successful termination
44
45   } // end main
```

```
Array b printed with:

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

Pointer/offset notation where the pointer is the array name
*(b + 0) = 10
*(b + 1) = 20
*(b + 2) = 30
*(b + 3) = 40
```

(Continued on top of next page)

Fig. 5.20 Referencing array elements with the array name and with pointers. (Part 2 of 3.)

(Continued from previous page)

```
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. 5.20 Referencing array elements with the array name and with pointers. (Part 3 of 3.)

To further illustrate the interchangeability of arrays and pointers, let us look at the two string copying functions—**copy1** and **copy2**—in the program of Fig. 5.21. Both functions copy a string into a character array. After a comparison of the function prototypes for **copy1** and **copy2**, the functions appear identical (because of the interchangeability of arrays and pointers). These functions accomplish the same task, but they are implemented differently.

```
1  // Fig. 5.21: fig05_21.cpp
2  // Copying a string using array notation
3  // and pointer notation.
4  #include <iostream>
5
6  using std::cout;
7  using std::endl;
8
9  void copy1( char *, const char * );  // prototype
10 void copy2( char *, const char * );  // prototype
11
12 int main()
13 {
14    char string1[ 10 ];
15    char *string2 = "Hello";
16    char string3[ 10 ];
17    char string4[] = "Good Bye";
18
19    copy1( string1, string2 );
20    cout << "string1 = " << string1 << endl;
21
22    copy2( string3, string4 );
23    cout << "string3 = " << string3 << endl;
24
25    return 0;  // indicates successful termination
26
27 } // end main
```

Fig. 5.21 String copying using array notation and pointer notation. (Part 1 of 2.)

```
28
29 // copy s2 to s1 using array notation
30 void copy1( char *s1, const char *s2 )
31 {
32    for ( int i = 0; ( s1[ i ] = s2[ i ] ) != '\0'; i++ )
33       ;   // do nothing in body
34
35 } // end function copy1
36
37 // copy s2 to s1 using pointer notation
38 void copy2( char *s1, const char *s2 )
39 {
40    for ( ; ( *s1 = *s2 ) != '\0'; s1++, s2++ )
41       ;   // do nothing in body
42
43 } // end function copy2
```

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

Fig. 5.21 String copying using array notation and pointer notation. (Part 2 of 2.)

Function **copy1** (lines 30–35) uses array subscript notation to copy the string in **s2** to the character array **s1**. The function declares an integer counter variable **i** to use as the array subscript. The **for** structure header (line 32) 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 condition in the **for**, **(s1[i] = s2[i]) != '\0'**, performs the copy operation character by character from **s2** to **s1**. When the null character is encountered in **s2**, it is assigned to **s1**, and the loop terminates, because the null character is equal to **'\0'**. Remember that the value of an assignment statement is the value assigned to its left operand.

Function **copy2** (lines 38–43) uses pointers and pointer arithmetic to copy the string in **s2** to the character array **s1**. Again, the **for** structure header (line 40) performs the entire copy operation. The header does not include any variable initialization. As in function **copy1**, the condition **(*s1 = *s2) != '\0'** performs the copy operation. Pointer **s2** is dereferenced, and the resulting character is assigned to the dereferenced pointer **s1**. After the assignment in the condition, the loop increments both pointers, so they point to the next element of array **s1** and the next character of string **s2**, respectively. When the loop encounters the null character in **s2**, the null character is assigned to the dereferenced pointer **s1** and the loop terminates. Note that the "increment portion" of this **for** structure has two increment expressions separated by a comma operator.

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 beyond the bounds of the array. Also, note that the second parameter of each function is declared as **const char *** (a pointer to a character constant—i.e., a constant string). In both functions, the second argument is copied into the first argument—characters are copied from the second argument one at a time, but the characters are never modified. Therefore, the second parameter is declared to point to a constant

value to enforce the principle of least privilege—neither function needs to modify the second argument, so neither function is allowed to modify the second argument.

5.9 Arrays of Pointers

Arrays may contain pointers. A common use of such a data structure 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 declaration of string array **suit** that might be useful in representing a deck of cards:

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

The **suit[4]** portion of the declaration indicates an array of four elements. The **char *** portion of the declaration indicates that each element of array **suit** is of type "pointer to **char**." The four values to be placed in the array are **"Hearts"**, **"Diamonds"**, **"Clubs"** and **"Spades"**. Each of these is stored in memory as a null-terminated character string that is one character longer than the number of characters between quotes. The four strings are seven, nine, six and seven 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, as shown in Fig. 5.22. 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.

The suit strings could be placed into a double-subscripted array in which each row represents one suit and each column represents one of the letters of a suit name. Such a data structure must have a fixed number of columns per row, and that number must be as large as the largest string. Therefore, considerable memory is wasted when a large number of strings is stored with most strings shorter than the longest string. We use arrays of strings to help represent a deck of cards in the next section.

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

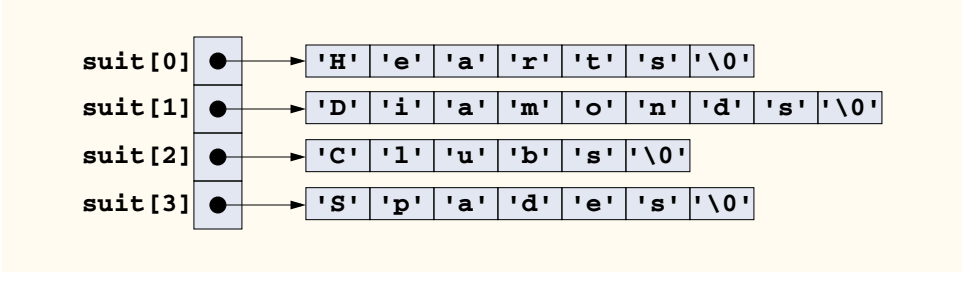

Fig. 5.22 Graphical representation of the **suit** array.

arguments is to pass options to a program. For example, from the command line on a Windows computer, the user can type

```
dir /P
```

to list the contents of the current directory and pause after each screen of information. When the **dir** command executes, the option **/P** is passed to **dir** as a command-line argument. Such arguments are placed in a string array that **main** receives as an argument. We discuss command-line arguments in Section 20.4.

5.10 Case Study: Card Shuffling and Dealing Simulation

This section uses 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 the exercises, 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. The top-down approach is particularly useful in attacking larger, more complex problems than we have seen in the early chapters.

We use a 4-by-13 double-subscripted array **deck** to represent the deck of playing cards (Fig. 5.23). 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—columns 0 through 9 correspond to faces ace through 10, respectively, and columns 10 through 12 correspond to jack, queen and king, respectively. We shall load string array **suit** with character strings representing the four suits and string array **face** with character strings representing the 13 face values.

This simulated deck of cards may be shuffled as follows. First the array **deck** is initialized 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

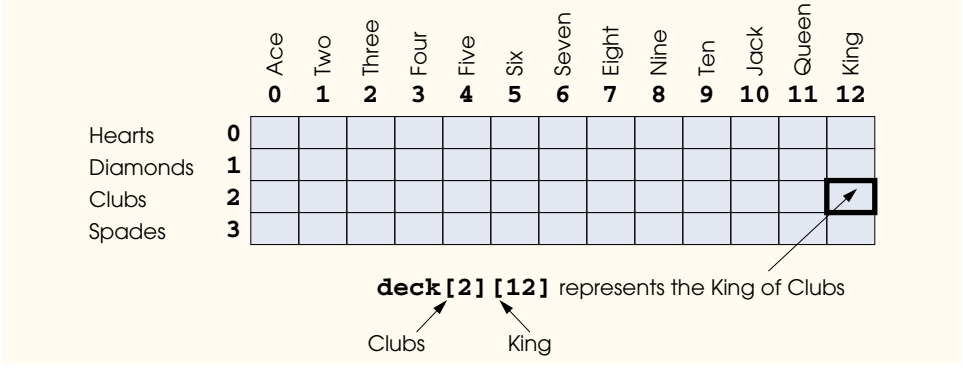

Fig. 5.23 Double-subscripted array representation of a deck of cards.

to be placed second, third, …, and 52nd in the shuffled deck. As the **deck** array begins to fill with card numbers, it is possible that a card will be selected twice (i.e., **deck[row][column]** will be nonzero when it is selected). This selection is simply ignored, and other **row**s and **column**s 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.

This shuffling algorithm could execute for an indefinitely long period 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 5.3

Sometimes algorithms that emerge in a "natural" way can contain subtle performance problems such as indefinite postponement. Seek algorithms that avoid indefinite postponement.

To deal the first card, we search the array for **deck[row][column]** matching **1**. This is accomplished with a nested **for** structure that varies **row** from 0 to 3 and **column** from 0 to 12. What card does that slot 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 "**. 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 us 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

"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

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

"Place card number in randomly selected unoccupied slot of deck" may be expanded as follows:

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 follows:

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

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

This completes the refinement process. Figure 5.24 contains the card shuffling and dealing program and a sample execution. Note the output formatting (lines 81–84) used in function **deal**. The output statement causes the face to be output right justified in a field of five characters and the suit to be output left justified in a field of eight characters. The output is printed in two-column format—if the card being output is in the first column (line 84), a tab is output after the card to move to the second column; otherwise, a newline is output.

```
1  // Fig. 5.24: fig05_24.cpp
2  // Card shuffling dealing program.
3  #include <iostream>
4
5  using std::cout;
6  using std::left;
7  using std::right;
8
9  #include <iomanip>
10
11 using std::setw;
12
```

Fig. 5.24 Card shuffling and dealing program. (Part 1 of 3.)

```
13 #include <cstdlib>  // prototypes for rand and srand
14 #include <ctime>    // prototype for time
15
16 // prototypes
17 void shuffle( int [][ 13 ] );
18 void deal( const int [][ 13 ], const char *[], const char *[] );
19
20 int main()
21 {
22    // initialize suit array
23    const char *suit[ 4 ] =
24       { "Hearts", "Diamonds", "Clubs", "Spades" };
25
26    // initialize face array
27    const char *face[ 13 ] =
28       { "Ace", "Deuce", "Three", "Four",
29         "Five", "Six", "Seven", "Eight",
30         "Nine", "Ten", "Jack", "Queen", "King" };
31
32    // initialize deck array
33    int deck[ 4 ][ 13 ] = { 0 };
34
35    srand( time( 0 ) );         // seed random-number generator
36
37    shuffle( deck );
38    deal( deck, face, suit );
39
40    return 0;  // indicates successful termination
41
42 } // end main
43
44 // shuffle cards in deck
45 void shuffle( int wDeck[][ 13 ] )
46 {
47    int row;
48    int column;
49
50    // for each of the 52 cards, choose slot of deck randomly
51    for ( int card = 1; card <= 52; card++ ) {
52
53       // choose new random location until unoccupied slot found
54       do {
55          row = rand() % 4;
56          column = rand() % 13;
57       } while ( wDeck[ row ][ column ] != 0 ); // end do/while
58
59       // place card number in chosen slot of deck
60       wDeck[ row ][ column ] = card;
61
62    } // end for
63
64 } // end function shuffle
```

Fig. 5.24 Card shuffling and dealing program. (Part 2 of 3.)

```
65
66  // deal cards in deck
67  void deal( const int wDeck[][ 13 ], const char *wFace[],
68             const char *wSuit[] )
69  {
70     // for each of the 52 cards
71     for ( int card = 1; card <= 52; card++ )
72
73        // loop through rows of wDeck
74        for ( int row = 0; row <= 3; row++ )
75
76           // loop through columns of wDeck for current row
77           for ( int column = 0; column <= 12; column++ )
78
79              // if slot contains current card, display card
80              if ( wDeck[ row ][ column ] == card ) {
81                 cout << setw( 5 ) << right << wFace[ column ]
82                      << " of " << setw( 8 ) << left
83                      << wSuit[ row ]
84                      << ( card % 2 == 0 ? '\n' : '\t' );
85
86              } // end if
87
88  } // end function deal
```

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

Fig. 5.24 Card shuffling and dealing program. (Part 3 of 3.)

There is also a weakness in the dealing algorithm. Once a match is found, even if it is found on the first try, the two inner **for** structures continue searching the remaining elements of **deck** for a match. In the exercises, we correct this deficiency.

5.11 Function Pointers

A pointer to a function contains the address of the function in memory. In Chapter 4, 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.

Multipurpose Bubble Sort Using Function Pointers

To illustrate the use of pointers to functions, Fig. 5.25 modifies the bubble sort program of Fig. 5.15. Figure 5.25 consists of **main** (lines 19–57) and the functions **bubble** (lines 61–74), **swap** (lines 78–84), **ascending** (lines 88–92) and **descending** (lines 96–100). Function **bubble** 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. Functions **ascending** and **descending** determine the sorting order. The program prompts the user to choose whether the array should be sorted in ascending order or in descending order. If the user enters 1, a pointer to function **ascending** is passed to function **bubble** (line 38), causing the array to be sorted into increasing order. If the user enters 2, a pointer to function **descending** is passed to function **bubble** (line 45), causing the array to be sorted into decreasing order.

```
1  // Fig. 5.25: fig05_25.cpp
2  // Multipurpose sorting program using function pointers.
3  #include <iostream>
4
5  using std::cout;
6  using std::cin;
7  using std::endl;
8
9  #include <iomanip>
10
11 using std::setw;
12
13 // prototypes
14 void bubble( int [], const int, bool (*)( int, int ) );
15 void swap( int * const, int * const );
16 bool ascending( int, int );
17 bool descending( int, int );
18
19 int main()
20 {
21    const int arraySize = 10;
22    int order;
23    int counter;
24    int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
```

Fig. 5.25 Multipurpose sorting program using function pointers. (Part 1 of 3.)

```
25
26     cout << "Enter 1 to sort in ascending order,\n"
27          << "Enter 2 to sort in descending order: ";
28     cin >> order;
29     cout << "\nData items in original order\n";
30
31     // output original array
32     for ( counter = 0; counter < arraySize; counter++ )
33        cout << setw( 4 ) << a[ counter ];
34
35     // sort array in ascending order; pass function ascending
36     // as an argument to specify ascending sorting order
37     if ( order == 1 ) {
38        bubble( a, arraySize, ascending );
39        cout << "\nData items in ascending order\n";
40     }
41
42     // sort array in descending order; pass function descending
43     // as an agrument to specify descending sorting order
44     else {
45        bubble( a, arraySize, descending );
46        cout << "\nData items in descending order\n";
47     }
48
49     // output sorted array
50     for ( counter = 0; counter < arraySize; counter++ )
51        cout << setw( 4 ) << a[ counter ];
52
53     cout << endl;
54
55     return 0;  // indicates successful termination
56
57  } // end main
58
59  // multipurpose bubble sort; parameter compare is a pointer to
60  // the comparison function that determines sorting order
61  void bubble( int work[], const int size,
62               bool (*compare)( int, int ) )
63  {
64     // loop to control passes
65     for ( int pass = 1; pass < size; pass++ )
66
67        // loop to control number of comparisons per pass
68        for ( int count = 0; count < size - 1; count++ )
69
70           // if adjacent elements are out of order, swap them
71           if ( (*compare)( work[ count ], work[ count + 1 ] ) )
72              swap( &work[ count ], &work[ count + 1 ] );
73
74  } // end function bubble
75
```

Fig. 5.25 Multipurpose sorting program using function pointers. (Part 2 of 3.)

```
76  // swap values at memory locations to which
77  // element1Ptr and element2Ptr point
78  void swap( int * const element1Ptr, int * const element2Ptr )
79  {
80     int hold = *element1Ptr;
81     *element1Ptr = *element2Ptr;
82     *element2Ptr = hold;
83
84  } // end function swap
85
86  // determine whether elements are out of order
87  // for an ascending order sort
88  bool ascending( int a, int b )
89  {
90     return b < a;   // swap if b is less than a
91
92  } // end function ascending
93
94  // determine whether elements are out of order
95  // for a descending order sort
96  bool descending( int a, int b )
97  {
98     return b > a;   // swap if b is greater than a
99
100 } // end function descending
```

```
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. 5.25 Multipurpose sorting program using function pointers. (Part 3 of 3.)

The following parameter appears in the function header for **bubble**:

```
bool ( *compare )( int, int )
```

This tells **bubble** to expect a parameter that is a pointer to a function that receives two integer parameters and returns a **bool** result. Parentheses are needed around ***compare** to indicate that **compare** is a pointer to a function. If we had not included the parentheses, the declaration would have been

```
bool *compare( int, int )
```

which declares a function that receives two integers as parameters and returns a pointer to a **bool** value.

The corresponding parameter in the function prototype of **bubble** is

```
bool (*)( int, int )
```

Note that only types have been included. However, for documentation purposes, the programmer can include names that the compiler will ignore.

The function passed to **bubble** is called in line 71 as follows:

```
( *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 execute the function.

The call to the function could have been made without dereferencing the pointer, as in

```
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 the name of an actual function in the program. This may be confusing to a user of the program who would like to see the definition of function **compare** and finds that it is never defined in the file.

Arrays of Pointers to Functions

One use of function pointers is in menu-driven systems. The program prompts a user to select an option (e.g., from 1 to 5) from a menu. Each option is serviced by a different function. Pointers to each function are stored in an array of pointers to functions. In this case, all the functions to which the array points must have the same return type and same parameter types. The user's choice is used as a subscript into the array of function pointers, and the pointer in the array is used to call the function.

Figure 5.26 provides a generic example of the mechanics of declaring and using an array of pointers to functions. Three functions are defined—**function1**, **function2** and **function3**—that each take an integer argument and do not return a value. Line 18 stores pointers to these three functions in array **f**. The declaration is read beginning in the leftmost set of parentheses as, "**f** is an array of three 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 (which, again, are pointers). When the user enters a value between 0 and 2, the value is used as the subscript into the array of pointers to functions. Line 30 invokes one of the functions in array **f**. In the call, **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 indicate that the function is called correctly. In the exercises, you will develop a menu-driven system.

```
1  // Fig. 5.26: fig05_26.cpp
2  // Demonstrating an array of pointers to functions.
```

Fig. 5.26 Array of pointers to functions. (Part 1 of 3.)

```
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

// function prototypes
void function1( int );
void function2( int );
void function3( int );

int main()
{
   // initialize array of 3 pointers to functions that each
   // take an int argument and return void
   void (*f[ 3 ])( int ) = { function1, function2, function3 };

   int choice;

   cout << "Enter a number between 0 and 2, 3 to end: ";
   cin >> choice;

   // process user's choice
   while ( choice >= 0 && choice < 3 ) {

      // invoke function at location choice in array f
      // and pass choice as an argument
      (*f[ choice ])( choice );

      cout << "Enter a number between 0 and 2, 3 to end: ";
      cin >> choice;
   }

   cout << "Program execution completed." << endl;

   return 0;  // indicates successful termination

} // end main

void function1( int a )
{
   cout << "You entered " << a
        << " so function1 was called\n\n";

} // end function1

void function2( int b )
{
   cout << "You entered " << b
        << " so function2 was called\n\n";

} // end function2

```

Fig. 5.26 Array of pointers to functions. (Part 2 of 3.)

```
56  void function3( int c )
57  {
58     cout << "You entered " << c
59          << " so function3 was called\n\n";
60
61  } // 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. 5.26 Array of pointers to functions. (Part 3 of 3.)

5.12 Introduction to Character and String Processing

In this section, we introduce some common standard library functions that facilitate string processing. The techniques discussed here are appropriate for developing text editors, word processors, page layout software, computerized typesetting systems and other kinds of text-processing software. We use pointer-based strings here. Chapter 8 introduces strings as full-fledged objects, and Chapter 15 explains strings as full-fledged objects in detail.

5.12.1 Fundamentals of Characters and Strings

Characters are the fundamental building blocks of C++ source programs. Every program is composed of a sequence of characters that—when grouped together meaningfully—is interpreted by the compiler as a series of instructions used to accomplish a task. A program may contain *character constants.* A character constant is 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. For example, **`'z'`** represents the integer value of **`z`** (122 in the ASCII character set; see Appendix B), and **`'\n'`** represents the integer value of newline (10 in the ASCII character set).

A string is a series of characters treated as a single unit. A string may include letters, digits and various *special characters* such as **`+`**, **`-`**, **`*`**, **`/`** and **`$`**. *String literals,* or *string constants,* in C++ are written in double quotation marks as follows:

`"John Q. Doe"`	(a name)
`"9999 Main Street"`	(a street address)
`"Maynard, Massachusetts"`	(a city and state)
`"(201) 555-1212"`	(a telephone number)

A string in C++ is an array of characters ending in the *null character* (**`'\0'`**), which specifies where the string terminates in memory. A string is accessed via a pointer to the first character in the string. The value of a string is the (constant) address of its first char-

acter. Thus, in C++, it is appropriate to say that *a string is a constant pointer*—in fact, a pointer to the string's first character. In this sense, strings are like arrays, because an array name is also a (constant) pointer to its first element.

A string may be assigned in a declaration to either a character array or a variable of type **char ***. The declarations

```
char color[] = "blue";
const char *colorPtr = "blue";
```

each initialize a variable to the string **"blue"**. The first declaration creates a five-element array **color** containing the characters **'b'**, **'l'**, **'u'**, **'e'** and **'\0'**. The second declaration creates pointer variable **colorPtr** that points to the letter **b** in the string **"blue"** somewhere in memory.

Portability Tip 5.5

*When a variable of type **char *** is initialized with a string literal, some compilers may place the string in a location in memory where the string cannot be modified. If a string literal must be modified in a program, it should be stored in a character array to ensure modifiability on all systems.*

The declaration **char color[] = "blue";** could also be written

```
char color[] = { 'b', 'l', 'u', 'e', '\0' };
```

When declaring a character array to contain a string, the array must be large enough to store the string and its terminating null character. The preceding declaration determines the size of the array, based on the number of initializers provided in the initializer list.

Common Programming Error 5.15

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

Common Programming Error 5.16

Creating or using a "string" that does not contain a terminating null character is an error.

Testing and Debugging Tip 5.3

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 is to be stored, characters beyond the end of the array will overwrite data in memory following the array.

A string can be stored in an array using stream extraction with **cin**. For example, the following statement can be used to store a string to character array **word[20]**:

```
cin >> word;
```

The string entered by the user is stored in **word**. The preceding statement reads characters until a whitespace character or end-of-file indicator is encountered. Note that the string should be no longer than 19 characters to leave room for the terminating null character. The **setw** stream manipulator introduced in Chapter 2 can be used to ensure that the string read into **word** does not exceed the size of the array. For example, the statement

```
cin >> setw( 20 ) >> word;
```

specifies that **cin** should read a maximum of 19 characters into array **word** and save the 20th location in the array to store the terminating null character for the string. The **setw** stream manipulator applies only to the next value being input.

In some cases, it is desirable to input an entire line of text into an array. For this purpose, C++ provides the function **cin.getline**. The **cin.getline** function takes three arguments—a character array in which the line of text will be stored, a length and a delimiter character. For example, the program segment

```
char sentence[ 80 ];
cin.getline( sentence, 80, '\n' );
```

declares array **sentence** of 80 characters and reads a line of text from the keyboard into the array. 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 last character in the array is reserved for the terminating null character.) If the delimiter character is encountered, it is read and discarded. The third argument to **cin.getline** has **'\n'** as a default value, so the preceding function call could have been written as follows:

```
cin.getline( sentence, 80 );
```

Chapter 12, Stream Input/Output, provides a detailed discussion of **cin.getline** and other input/output functions.

Common Programming Error 5.17

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

Common Programming Error 5.18

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

5.12.2 String Manipulation Functions of the String-Handling Library

The string-handling library provides many useful functions for manipulating string data, comparing strings, searching strings for characters and other strings, tokenizing strings (separating strings into logical pieces) and determining the length of strings. This section presents some common string-manipulation functions of the string-handling library (from the C++ standard library). The functions are summarized in Fig. 5.27. The prototypes for these functions are located in header file **<cstring>**.

Note that several functions in Fig. 5.27 contain parameters with data type **size_t**. This type is defined in the header file **<cstring>** to be an unsigned integral type such as **unsigned int** or **unsigned long**.

Common Programming Error 5.19

*Forgetting to include the **<cstring>** header file when using functions from the string handling library causes compilation errors.*

Function prototype	Function description
`char *strcpy( char *s1, const char *s2 );`	Copies the string **s2** into the character array **s1**. The value of **s1** is returned.
`char *strncpy( char *s1, const char *s2, size_t n );`	Copies at most **n** characters of the string **s2** into the character array **s1**. The value of **s1** is returned.
`char *strcat( char *s1, const char *s2 );`	Appends the string **s2** to the string **s1**. The first character of **s2** overwrites the terminating null character of **s1**. The value of **s1** is returned.
`char *strncat( char *s1, const char *s2, size_t n );`	Appends at most **n** characters of string **s2** to string **s1**. The first character of **s2** overwrites the terminating null character of **s1**. The value of **s1** is returned.
`int strcmp( const char *s1, const char *s2 );`	Compares the string **s1** with the string **s2**. The function returns a value of zero, less than zero or greater than zero if **s1** is equal to, less than or greater than **s2**, respectively.
`int strncmp( const char *s1, const char *s2, size_t n );`	Compares up to **n** characters of the string **s1** with the string **s2**. The function returns zero, less than zero or greater than zero if the **n**-character portion of **s1** is equal to, less than or greater than the corresponding **n**-character portion of **s2**, respectively.
`char *strtok( char *s1, const char *s2 );`	A sequence of calls to **strtok** breaks string **s1** into "tokens"—logical pieces such as words in a line of text—delimited by characters contained in string **s2**. The first call contains **s1** as the first argument, and subsequent calls to continue tokenizing the same string contain **NULL** as the first argument. A pointer to the current token is returned by each call. If there are no more tokens when the function is called, **NULL** is returned.
`size_t strlen( const char *s );`	Determines the length of string **s**. The number of characters preceding the terminating null character is returned.

Fig. 5.27 String-manipulation functions of the string-handling library.

Copying Strings with ***strcpy*** *and* ***strncpy***

Function ***strcpy*** copies its second argument—a string—into its first argument—a character array that must be large enough to store the string and its terminating null character, (which is also copied). Function ***strncpy*** is equivalent to **strcpy**, except that **strncpy** specifies the number of characters to be copied from the string into the array. Note that function **strncpy** does not necessarily copy the terminating null character of its second argument—a terminating null character is written only if the number of characters

to be copied is at least one more than the length of the string. For example, if **"test"** is the second argument, a terminating null character is written only if the third argument to **strncpy** is at least **5** (four characters in **"test"** plus one terminating null character). If the third argument is larger than **5**, null characters are appended to the array until the total number of characters specified by the third argument is written.

Common Programming Error 5.20

*Not appending a terminating null character to the first argument of a **strncpy** (in a statement after the **strncpy** call) when the third argument is less than or equal to the length of the string in the second argument can cause fatal run-time errors.*

Figure 5.28 uses **strcpy** (line 16) to copy the entire string in array **x** into array **y** and uses **strncpy** (line 22) to copy the first **14** characters of array **x** into array **z**. Line 23 appends a null character (**'\0'**) to array **z**, because the call to **strncpy** in the program does not write a terminating null character. (The third argument is less than the string length of the second argument plus one.)

```
1  // Fig. 5.28: fig05_28.cpp
2  // Using strcpy and strncpy.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  #include <cstring>   // prototypes for strcpy and strncpy
9
10 int main()
11 {
12    char x[] = "Happy Birthday to You";
13    char y[ 25 ];
14    char z[ 15 ];
15
16    strcpy( y, x );  // copy contents of x into y
17
18    cout << "The string in array x is: " << x
19         << "\nThe string in array y is: " << y << '\n';
20
21    // copy first 14 characters of x into z
22    strncpy( z, x, 14 );  // does not copy null character
23    z[ 14 ] = '\0';       // append '\0' to z's contents
24
25    cout << "The string in array z is: " << z << endl;
26
27    return 0;  // indicates successful termination
28
29 } // end main
```

```
The string in array x is: Happy Birthday to You
The string in array y is: Happy Birthday to You
The string in array z is: Happy Birthday
```

Fig. 5.28 **strcpy** and **strncpy**.

Concatenating Strings with `strcat` *and* `strncat`

Function **`strcat`** appends its second argument (a string) to its first argument (a character array containing a string). The first character of the second argument replaces the null character (`'\0'`) that terminates the string in the first argument. The programmer must ensure that the array used to store the first string is large enough to store the combination of the first string, the second string and the terminating null character (copied from the second string). Function **`strncat`** appends a specified number of characters from the second string to the first string and appends a terminating null character to the result. The program of Fig. 5.29 demonstrates function **`strcat`** (lines 18 and 29) and function **`strncat`** (line 24).

Comparing Strings with `strcmp` *and* `strncmp`

Figure 5.30 compares three strings using **`strcmp`** (lines 22, 24 and 25) and **`strncmp`** (lines 28, 29 and 31). Function **`strcmp`** compares its first string argument with its second string argument character by character. The function returns zero if the strings are equal, a

```
1  // Fig. 5.29: fig05_29.cpp
2  // Using strcat and strncat.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  #include <cstring>   // prototypes for strcat and strncat
9
10 int main()
11 {
12    char s1[ 20 ] = "Happy ";
13    char s2[] = "New Year ";
14    char s3[ 40 ] = "";
15
16    cout << "s1 = " << s1 << "\ns2 = " << s2;
17
18    strcat( s1, s2 );  // concatenate s2 to s1
19
20    cout << "\n\nAfter strcat(s1, s2):\ns1 = " << s1
21         << "\ns2 = " << s2;
22
23    // concatenate first 6 characters of s1 to s3
24    strncat( s3, s1, 6 );  // places '\0' after last character
25
26    cout << "\n\nAfter strncat(s3, s1, 6):\ns1 = " << s1
27         << "\ns3 = " << s3;
28
29    strcat( s3, s1 );  // concatenate s1 to s3
30    cout << "\n\nAfter strcat(s3, s1):\ns1 = " << s1
31         << "\ns3 = " << s3 << endl;
32
33    return 0;  // indicates successful termination
34
35 } // end main
```

Fig. 5.29 **`strcat`** and **`strncat`**. (Part 1 of 2.)

```
s1 = Happy
s2 = New Year

After strcat(s1, s2):
s1 = Happy New Year
s2 = New Year

After strncat(s3, s1, 6):
s1 = Happy New Year
s3 = Happy

After strcat(s3, s1):
s1 = Happy New Year
s3 = Happy Happy New Year
```

Fig. 5.29 **strcat** and **strncat**. (Part 2 of 2.)

negative value if the first string is less than the second string and a positive value if the first string is greater than the second string. Function **strncmp** is equivalent to **strcmp**, except that **strncmp** compares up to a specified number of characters. Function **strncmp** stops comparing characters if it reaches the null character in one of its string arguments. The program prints the integer value returned by each function call.

```
1  // Fig. 5.30: fig05_30.cpp
2  // Using strcmp and strncmp.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  #include <iomanip>
9
10 using std::setw;
11
12 #include <cstring>  // prototypes for strcmp and strncmp
13
14 int main()
15 {
16    char *s1 = "Happy New Year";
17    char *s2 = "Happy New Year";
18    char *s3 = "Happy Holidays";
19
20    cout << "s1 = " << s1 << "\ns2 = " << s2
21         << "\ns3 = " << s3 << "\n\nstrcmp(s1, s2) = "
22         << setw( 2 ) << strcmp( s1, s2 )
23         << "\nstrcmp(s1, s3) = " << setw( 2 )
24         << strcmp( s1, s3 ) << "\nstrcmp(s3, s1) = "
25         << setw( 2 ) << strcmp( s3, s1 );
26
```

Fig. 5.30 **strcmp** and **strncmp**. (Part 1 of 2.)

```
27      cout << "\n\nstrncmp(s1, s3, 6) = " << setw( 2 )
28           << strncmp( s1, s3, 6 ) << "\nstrncmp(s1, s3, 7) = "
29           << setw( 2 ) << strncmp( s1, s3, 7 )
30           << "\nstrncmp(s3, s1, 7) = "
31           << setw( 2 ) << strncmp( s3, s1, 7 ) << endl;
32
33      return 0;  // indicates successful termination
34
35   } // end main
```

```
s1 = Happy New Year
s2 = Happy New Year
s3 = Happy Holidays

strcmp(s1, s2) =  0
strcmp(s1, s3) =  1
strcmp(s3, s1) = -1

strncmp(s1, s3, 6) =  0
strncmp(s1, s3, 7) =  1
strncmp(s3, s1, 7) = -1
```

Fig. 5.30 **`strcmp`** and **`strncmp`**. (Part 2 of 2.)

Common Programming Error 5.21

Assuming that ***`strcmp`*** *and* ***`strncmp`*** *return one (a true value) when their arguments are equal is a logic error. Both functions return zero (C++'s false value) for equality. Therefore, when testing two strings for equality, the result of the* ***`strcmp`*** *or* ***`strncmp`*** *function should be compared with zero to determine whether the strings are equal.*

To understand just what it means for one string to be "greater than" or "less than" another string, consider the process of alphabetizing a series of last names. The reader would, no doubt, place "Jones" before "Smith," because the first letter of "Jones" comes before the first letter of "Smith" in the alphabet. But the alphabet is more than just a list of 26 letters—it is an ordered list of characters. Each letter occurs in a specific position within the list. "Z" is more than just a letter of the alphabet; "Z" is specifically the 26th letter of the alphabet.

How does the computer know that one letter comes before another? All characters are represented inside the computer as numeric codes; when the computer compares two strings, it actually compares the numeric codes of the characters in the strings.

In an effort to standardize character representations, most computer manufacturers have designed their machines to utilize one of two popular coding schemes—*ASCII* or *EBCDIC*. ASCII stands for "American Standard Code for Information Interchange," and EBCDIC stands for "Extended Binary Coded Decimal Interchange Code." There are other coding schemes, but these two are the most popular.

ASCII and EBCDIC are called *character codes,* or *character sets.* Most readers of this book will be using desktop or notebook computers that use the ASCII character set. IBM mainframe computers use the EBCDIC character set. As Internet and World Wide Web usage becomes pervasive, the newer Unicode character set is growing rapidly in popularity. For more information on Unicode, visit **`www.unicode.org`**. String and character manipula-

tions actually involve the manipulation of the appropriate numeric codes and not the characters themselves. This explains the interchangeability of characters and small integers in C++. Since it is meaningful to say that one numeric code is greater than, less than or equal to another numeric code, it becomes possible to relate various characters or strings to one another by referring to the character codes. Appendix B contains the ASCII character codes.

Portability Tip 5.6

The internal numeric codes used to represent characters may be different on different computers, because these computers may use different character sets.

Portability Tip 5.7

Do not explicitly test for ASCII codes, as in **`if ( rating == 65 )`***; rather, use the corresponding character constant, as in* **`if ( rating == 'A' )`**.

Tokenizing a String with `strtok`

Function **`strtok`** breaks a string into a series of *tokens.* A token is a sequence of characters separated by *delimiting characters* (usually spaces or punctuation marks). For example, in a line of text, each word can be considered a token, and the spaces separating the words can be considered delimiters.

Multiple calls to **`strtok`** are required to break a string into tokens (assuming that the string contains more than one token). The first call to **`strtok`** contains two arguments, a string to be tokenized and a string containing characters that separate the tokens (i.e., delimiters). Line 19 in Fig. 5.31 assigns to **`tokenPtr`** a pointer to the first token in **`sentence`**. The second argument, **`" "`**, indicates that tokens in **`sentence`** are separated by spaces. Function **`strtok`** searches for the first character in **`sentence`** that is not a delimiting character (space). This begins the first token. The function then finds the next delimiting character in the string and replaces it with a null (**`'\0'`**) character. This terminates the current token. Function **`strtok`** saves a pointer to the next character following the token in **`sentence`** and returns a pointer to the current token.

```
1  // Fig. 5.31: fig05_31.cpp
2  // Using strtok.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  #include <cstring>  // prototype for strtok
9
10 int main()
11 {
12    char sentence[] = "This is a sentence with 7 tokens";
13    char *tokenPtr;
14
15    cout << "The string to be tokenized is:\n" << sentence
16         << "\n\nThe tokens are:\n\n";
17
```

Fig. 5.31 **`strtok`**. (Part 1 of 2.)

```
18      // begin tokenization of sentence
19      tokenPtr = strtok( sentence, " " );
20
21      // continue tokenizing sentence until tokenPtr becomes NULL
22      while ( tokenPtr != NULL ) {
23         cout << tokenPtr << '\n';
24         tokenPtr = strtok( NULL, " " );  // get next token
25
26      } // end while
27
28      cout << "\nAfter strtok, sentence = " << sentence << endl;
29
30      return 0;  // indicates successful termination
31
32   } // end main
```

```
The string to be tokenized is:
This is a sentence with 7 tokens

The tokens are:

This
is
a
sentence
with
7
tokens

After strtok, sentence = This
```

Fig. 5.31 **strtok**. (Part 2 of 2.)

Subsequent calls to **strtok** to continue tokenizing **sentence** contain **NULL** as the first argument (line 24). The **NULL** argument indicates that the call to **strtok** should continue tokenizing from the location in **sentence** saved by the last call to **strtok**. Note that **strtok** maintains this saved information in a manner that is not visible to the programmer. If no tokens remain when **strtok** is called, **strtok** returns **NULL**. The program of Fig. 5.31 uses **strtok** to tokenize the string **"This is a sentence with 7 tokens"**. The program prints each token on a separate line. Line 28 outputs **sentence** after tokenization. Note that **strtok** modifies the input string; therefore, a copy of the string should be made if the program requires the original after the calls to **strtok**. When **sentence** is output after tokenization, note that only the word "**This**" prints, because **strtok** replaced each blank in **sentence** with a null character (**'\0'**) during the tokenization process.

Common Programming Error 5.22

*Not realizing that **strtok** modifies the string being tokenized and then attempting to use that string as if it were the original unmodified string is a logic error.*

Determining String Lengths

Function **strlen** takes a string as an argument and returns the number of characters in the string—the terminating null character is not included in the length. The program of Fig. 5.32 demonstrates function **strlen**.

```
1  // Fig. 5.32: fig05_32.cpp
2  // Using strlen.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  #include <cstring>  // prototype for strlen
9
10 int main()
11 {
12    char *string1 = "abcdefghijklmnopqrstuvwxyz";
13    char *string2 = "four";
14    char *string3 = "Boston";
15
16    cout << "The length of \"" << string1
17         << "\" is " << strlen( string1 )
18         << "\nThe length of \"" << string2
19         << "\" is " << strlen( string2 )
20         << "\nThe length of \"" << string3
21         << "\" is " << strlen( string3 ) << endl;
22
23    return 0;  // indicates successful termination
24
25 } // end main
```

```
The length of "abcdefghijklmnopqrstuvwxyz" is 26
The length of "four" is 4
The length of "Boston" is 6
```

Fig. 5.32 `strlen`.

5.13 (Optional Case Study) Thinking About Objects: Collaborations Among Objects

This is the last of our object-oriented design sections before we begin our study of C++ object-oriented programming in Chapter 6. After we discuss the interactions among objects in this section and discuss creating classes and objects in Chapter 6, we begin coding the elevator simulator in C++. To complete the elevator simulator, we also use the C++ techniques discussed in Chapter 7 and Chapter 9. We have included at the end of this section a list of Internet and World Wide Web UML resources and a bibliography of UML references.

In the "Thinking About Objects" section at the end of Chapter 4, we began to investigate how objects interact by discussing how a **`Scheduler`** object interacts with other objects to schedule a person to step onto a floor. In this section, we concentrate on the interactions among other objects in the system. When two or more objects communicate with one another to accomplish a task, they interact with one another by sending and receiving messages.

When two objects interact, a message sent by one object invokes an operation of the second object (just as pressing down the accelerator pedal in a car signals the car to "go faster" and pressing the brake pedal signals the car to "go slower"). In the "Thinking About Objects" section at the end of Chapter 4, we determined many of the operations of the classes in our system. In this section, we concentrate on the messages that invoke these operations.

Figure 5.33 is the table of classes and verb phrases from Section 4.10. We removed all the verb phrases that do not correspond to a message sent between two objects. (For example, we eliminate "moves" in the **`Elevator`** because the **`Elevator`** sends this message to itself.) The remaining phrases are likely to correspond to the interactions between objects in our system. We associate the phrases "provides the time to the scheduler" and "provides the time to the elevator" with class **`Building`**, because we decided in Chapter 4 that the building controls the simulation. We associate the phrases "increments the clock time" and "gets the time from the clock" with class **`Building`** for the same reason.

We examine the list of verbs to determine the interactions in our system. For example, class **`Elevator`** lists the phrase "resets the elevator button." To accomplish this task, an object of class **`Elevator`** must send the **`resetButton`** message to an object of class **`ElevatorButton`**, invoking the **`resetButton`** operation of that class. Figure 5.34 lists all the interactions that can be gleaned from our table of verb phrases.

Class	Verb phrases
`Elevator`	resets the elevator button, sounds the elevator bell, signals its arrival to a floor, opens its door, closes its door
`Clock`	none in problem statement
`Scheduler`	verifies that a floor is unoccupied
`Person`	steps onto a floor, presses floor button, presses elevator button, enters elevator, exits elevator
`Floor`	resets floor button, turns off light, turns on light
`FloorButton`	summons elevator
`ElevatorButton`	signals elevator to prepare to leave
`Door`	(opening of door) signals person to exit elevator, (opening of door) signals person to enter elevator
`Bell`	none in problem statement
`Light`	none in problem statement
`Building`	increments the clock time, gets the time from the clock, provides the time to the scheduler, provides the time to the elevator

Fig. 5.33 Modified list of verb phrases for classes in the system.

An object of class	Sends the message	To an object of class
`Elevator`	**`resetButton`**	**`ElevatorButton`**
	`ringBell`	**`Bell`**
	`elevatorArrived`	**`Floor`**
	`openDoor`	**`Door`**
	`closeDoor`	**`Door`**

Fig. 5.34 Collaborations that occur in the elevator system. (Part 1 of 2.)

An object of class	Sends the message	To an object of class
Clock	-----	-----
Scheduler	stepOntoFloor	Person
	isOccupied	Floor
Person	pressButton	FloorButton
	pressButton	ElevatorButton
	passengerEnters	Elevator
	passengerExits	Elevator
	personArrives	Floor
Floor	resetButton	FloorButton
	turnOff	Light
	turnOn	Light
FloorButton	summonElevator	Elevator
ElevatorButton	prepareToLeave	Elevator
Door	exitElevator	Person
	enterElevator	Person
Bell	-----	-----
Light	-----	-----
Building	tick	Clock
	getTime	Clock
	processTime	Scheduler
	processTime	Elevator

Fig. 5.34 Collaborations that occur in the elevator system. (Part 2 of 2.)

Collaboration Diagrams

Now let us consider the objects that must interact so that people in our simulation can enter and exit the elevator when it arrives on a floor. A *collaboration* consists of a collection of objects that work together to perform a task. The UML enables us to model such objects, and their interactions, with *collaboration diagrams*. Collaboration diagrams and sequence diagrams both provide information about how objects interact, but each diagram emphasizes different information. Sequence diagrams emphasize *when* interactions occur. Collaboration diagrams emphasize *which objects participate* in the interactions and the relationships among those objects.

Figure 5.35 shows a collaboration diagram that models the interaction among objects in our system as objects of class **`Person`** enter and exit the elevator. The collaboration begins when the elevator arrives on a floor. As in a sequence diagram, an object in a collaboration diagram is represented as a rectangle that encloses the object's name.

Interacting objects are connected with solid lines, and objects pass messages to one another along these lines in the direction shown by the arrows. Each message's name and a *message number* appear next to the corresponding arrow.

The *sequence of messages* in a collaboration diagram progresses in numerical order from least to greatest. In this diagram, the numbering starts with message **`1`**. When the elevator

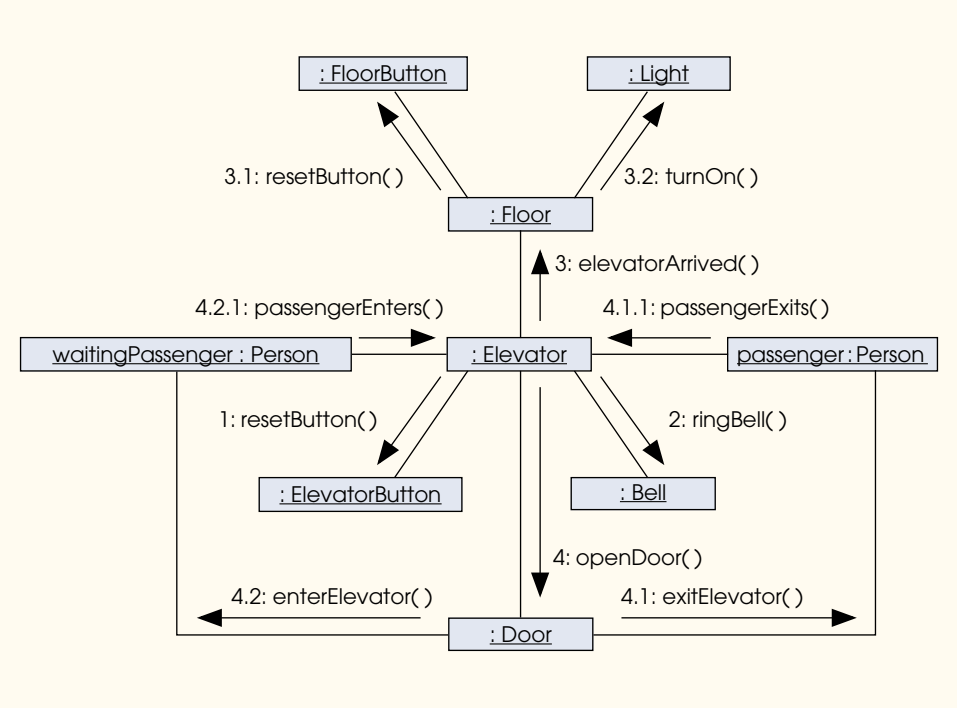

Fig. 5.35 Collaboration diagram for loading and unloading passengers.

arrives at a floor, the first thing it does is send this message (**`resetButton`**) to the elevator button to reset the button. The elevator then sends the **`ringBell`** message (message 2) to the bell. Then the elevator notifies the floor of its arrival (message 3), so that the floor can reset its button and turn on its light (messages 3.1 and 3.2, respectively).

After the floor has reset its button and turned on its light, the elevator opens its door (message 4). At this point, the door sends the **`exitElevator`** message (message 4.1) to the **`passenger`** object.[1] The **`passenger`** object notifies the elevator of its intent to exit via the **`passengerExits`** message (message 4.1.1).

After the person riding the elevator has exited, the person waiting on the floor (the **`waitingPassenger`** object) can enter the elevator. Notice that the door sends the **`enterElevator`** message (message 4.2) to the **`waitingPassenger`** object after the **`passenger`** object sends the **`passengerExits`** message to the elevator (message 4.1.1). This sequence ensures that a person on the floor waits for an elevator passenger to exit before the person on the floor enters the elevator. The **`waitingPassenger`** object enters the elevator via the **`passengerEnters`** message (message 4.2.1). Determining the sequence of these messages and modeling them with a diagram will aid us as we implement the various classes in our system.

1. In the real world, a person riding on the elevator waits until the door opens before exiting the elevator. We must model this behavior; therefore, we have the door send a message to the **`passenger`** object in the elevator. This message represents a visual cue to the person in the elevator. When the person receives the cue, the person can exit the elevator.

Summary

We now have a reasonably complete listing of the classes and objects to implement our elevator simulator, as well as the interactions among the objects of these classes. In the next chapter, we begin our study of object-oriented programming in C++. After reading Chapter 6, we will be ready to write a substantial portion of the elevator simulator in C++. After completing Chapter 7, we implement a complete, working elevator simulator. In Chapter 9, we discuss how to use inheritance to exploit commonality among classes to minimize the amount of software needed to implement a system.

Let us summarize our simplified object-oriented development process[2] that we have presented in Chapter 2–Chapter 5:

1. In the analysis phase, meet with the clients (the people who want you to build their system) and gather as much information as possible about the system. With this information, create the use cases that describe the ways in which users interact with the system. (In our case study, we do not concentrate on the analysis phase. The results of this phase are represented in the problem statement, and the use cases derive from this statement.) We note again that real-world systems often have many use cases. Throughout the remaining steps, we continually evaluate our design against the use cases to be sure that our end product matches the information we obtained from analyzing the system requirements.
2. Begin identifying the classes in the system by listing the nouns in the problem statement. Filter the list by eliminating nouns that represent obvious attributes of classes and other nouns that have no relevance to the software system being modeled. Create a class diagram that models the classes in the system and their relationships (associations).
3. Extract the attributes of each class from the problem statement by listing words and phrases that describe each class in the system.
4. Learn more about the dynamic nature of the system. Create statechart diagrams to learn how the objects in the system change over time.
5. Examine verbs and verb phrases associated with each class. Use these phrases to extract the operations of the classes in our system. Activity diagrams can help model the details of these operations.
6. Examine the interactions among various objects. Use sequence and collaboration diagrams to model these interactions. Add attributes and operations to the classes as the design process reveals the need for them.
7. At this point, our design probably still has a few missing pieces. These will become apparent as we begin to implement our elevator simulator in C++ in the "Thinking About Objects" section at the end of Chapter 6.

---

2. We created this basic OOD process to introduce readers to object-oriented design using the UML. Readers who wish to pursue this topic in more depth can study the more formal and detailed Rational Unified Process. For more information on this software-design methodology, we recommend reading *The Rational Unified Process: An Introduction (2nd Edition)* by Philippe Kruchten and *The Unified Software Develpment Process* by Ivar Jacobson, Grady Booch and James Rumbaugh. For online resources, visit **`www.therationaledge.com`**, which contains numerous articles on this process.

UML Resources on the Internet and World Wide Web

The following is a collection of Internet and World Wide Web resources for the UML. These include the UML 1.4 specification and other reference materials, general resources, tutorials, FAQs, articles, whitepapers and software.

References

www.omg.org
This is the Object Management Group (OMG) site. The OMG is responsible for overseeing maintenance and future revisions of the UML. The site contains information about the UML and other object-oriented technologies.

www.rational.com
Rational Software Corporation developed the UML. Its Web site contains information about the UML and the creators of the UML—Grady Booch, James Rumbaugh and Ivar Jacobson.

www.omg.org/technology/documents/formal/uml.htm
This site contains the official UML 1.4 specification.

www.rational.com/uml/resources/quick/index.jtmpl
Rational Software Corporation's UML quick-reference guide can be found at this site.

www.holub.com/class/uml/uml.html
This site provides a detailed UML quick-reference card with additional commentary.

softdocwiz.com/UML.htm
Kendall Scott, an author of several UML resources, maintains a UML dictionary at this site.

Resources

www.omg.org/uml/
This site contains the OMG UML resource page.

www.rational.com/uml/index.jsp
Rational Software Corporation's UML resource page

www.platinum.com/corp/uml/uml.htm
UML Partners member Platinum Technology maintains this UML resource site.

www.cetus-links.org/oo_uml.html
This site contains hundreds of links to other UML sites, including information, tutorials and software.

www.embarcadero.com/support/uml_central.asp
This site contains links to several UML-related items, including references, tutorials and articles.

www.devx.com/uml
This site contains a wealth of UML information, including articles and links to news groups and to other sites.

www.celigent.com/uml
This site contains general information and links to important sites on the Web.

www.methods-tools.com/cgi-bin/DiscussionUML.cgi
This site provides access to several UML discussion groups.

www.pols.co.uk/usecasezone/index.htm
This site provides resources and articles about applying use cases.

www.ics.uci.edu/pub/arch/uml/uml_books_and_tools.html
This site contains links to information about books on the UML, as well as a list of tools that support UML notation.

home.earthlink.net/~salhir
Sinan Si Alhir, author of *UML in a Nutshell*, maintains this site; it includes links to many UML resources.

www.rational.com/products/rup/index.jsp
This is the main site for the Rational Unified Process (RUP), Rational's OOAD methodology.

www.cetus-links.org/oo_ooa_ood_methods.html
This site contains links to many software development methodologies, including RUP, Extreme Programming, the Booch methodology and many more.

Software

www.rational.com/products/rose/index.jsp
This site is the home page for Rational Software Corporation's UML visual modeling tool, Rational Rose.™ You can download a trial version from this location and use it free for a limited time.

www.sparxsystems.com.au/ea.htm
Sparx Systems offers Enterprise Architect, a UML OOAD tool. The professional version provides code generation and reverse-engineering support for C++, Java and C#, among others.

www.visualobject.com
Visual Object Modelers has created a visual UML modeling tool. You can download a limited demonstration version from this Web site and use it free for a limited time.

www.embarcadero.com/downloads/download.asp
Embarcadero provides Desribe™ Enterprise, a UML design tool.

www.microgold.com/version2/stage/product.html
Microgold Software, Inc. has created *With*Class, a software design application that supports the UML notation.

dir.lycos.com/Computers/Software/Object_Oriented/Methodologies/UML/Tools
This site lists dozens of UML modeling tools and their home pages.

www.methods-tools.com/tools/modeling.html
This site contains a listing of many object modeling tools, including those that support the UML.

Articles and Whitepapers

www.omg.org/news/pr99/UML_2001_CACM_Oct99_p29-Kobryn.pdf
This article, written by Cris Kobryn, explores the past, present and future of the UML.

www.db.informatik.uni-bremen.de/umlbib
The UML Bibliography provides names and authors of many UML-related articles. You can search articles by author or title.

www.ratio.co.uk/white.html
You can read a whitepaper that outlines a process for OOAD using the UML at this site. The paper also includes some implementation in C++.

www.conallen.com/whitepapers/webapps/ModelingWebApplications.htm
This site contains a case study that models Web applications using the UML.

www.sdmagazine.com
The Software Development Magazine Online site has a repository of many articles on the UML. You can search by subject or browse article titles.

FAQs

www.rational.com/uml/gstart/faq.jsp
This is the location of Rational Software Corporation's UML FAQ.

www.jguru.com/faq
Enter UML in the search box to access a this site's UML FAQ.

Bibliography

Alhir, S. *UML in a Nutshell*. Cambridge: O'Reily & Associates, Inc., 1998.

Booch, G., Rumbaugh, J. and Jacobson, I. *The Unified Modeling Language User Guide*. Reading, MA: Addison-Wesley, 1999.

Firesmith, D.G., and Henderson-Sellers, B. "Clarifying Specialized Forms of Association in UML and OML." *Journal of Object-Oriented Programming* May 1998: 47–50.

Fowler, M., and Scott, K. *UML Distilled: Applying the Standard Object Modeling Language*. Reading, MA: Addison-Wesley, 1997.

Johnson, L.J. "Model Behavior." *Enterprise Development* May 2000: 20–28.

McLaughlin, M., and A. Moore. "Real-Time Extensions to the UML." *Dr. Dobb's Journal* December 1998: 82–93.

Melewski, D. "UML Gains Ground." *Application Development Trends* October 1998: 34–44.

Melewski, D. "UML: Ready for Prime Time?" *Application Development Trends* November 1997: 30–44.

Melewski, D. "Wherefore and what now, UML?" *Application Development Trends* December 1999: 61–68.

Muller, P. *Instant UML*. Birmingham, UK: Wrox Press Ltd, 1997.

Perry, P. "UML Steps to the Plate." *Application Development Trends* May 1999: 33–36.

Rumbaugh, J., Jacobson, I. and Booch, G. *The Unified Modeling Language Reference Manual*. Reading, MA: Addison-Wesley, 1999.

Schmuller, J. *Sam's Teach Yourself UML in 24 Hours*. Indianapolis: Macmillan Computer Publishing, 1999.

The Unified Modeling Language Specification: Version 1.4. Framingham, MA: Object Management Goup (OMG), 2001.

SUMMARY

- Pointers are variables that contain as their values addresses of other variables.
- The declaration

```
int *ptr;
```

  declares **ptr** to be a pointer to a variable of type **int** and is read, "**ptr** is a pointer to **int**." The ***** as used here in a declaration indicates that the variable is a pointer.
- There are three values that can be used to initialize a pointer: **0**, **NULL** or an address of an object of the same type. Initializing a pointer to **0** and initializing that same pointer to **NULL** are identical—**0** is the convention in C++.
- The only integer that can be assigned to a pointer without casting is zero.
- The **&** (address) operator returns the memory address of its operand.

- The operand of the address operator must be a variable name (or another *lvalue*); the address operator cannot be applied to constants or to expressions that do not return a reference.
- The ***** operator, referred to as the indirection (or dereferencing) operator, returns a synonym, alias or nickname for the name of the object that its operand points to in memory. This is called dereferencing the pointer.
- When calling a function with an argument that the caller wants the called function to modify, the address of the argument may be passed. The called function then uses the indirection operator (*****) to dereference the pointer and modify the value of the argument in the calling function.
- A function receiving an address as an argument must have a pointer as its corresponding parameter.
- The **const** qualifier enables the programmer to inform the compiler that the value of a particular variable cannot be modified through the specified identifier. If an attempt is made to modify a **const** value, the compiler issues either a warning or an error, depending on the particular compiler.
- There are four ways to pass a pointer to a function—a nonconstant pointer to nonconstant data, a nonconstant pointer to constant data, a constant pointer to nonconstant data and a constant pointer to constant data.
- The value of the array name is the address of (a pointer to) the array's first element.
- To pass a single element of an array by reference using pointers, pass the address of the specific array element.
- C++ provides unary operator **sizeof** to determine the size of an array (or of any other data type, variable or constant) in bytes at compile time.
- When applied to the name of an array, the **sizeof** operator returns the total number of bytes in the array as an integer.
- The arithmetic operations that may be performed on pointers are incrementing (**++**) a pointer, decrementing (**--**) a pointer, adding (**+** or **+=**) an integer to a pointer, subtracting (**-** or **-=**) an integer from a pointer and subtracting one pointer from another.
- When an integer is added or subtracted from a pointer, the pointer is incremented or decremented by that integer times the size of the object to which the pointer refers.
- Pointer arithmetic operations should only be performed on contiguous portions of memory such as an array. All elements of an array are stored contiguously in memory.
- Pointers can be assigned to one another if both pointers are of the same type. Otherwise, a cast must be used. The exception to this is a **void *** pointer, which is a generic pointer type that can hold pointer values of any type. Pointers to **void** can be assigned pointers of other types. A **void *** pointer can be assigned to a pointer of another type only with an explicit type cast.
- The only valid operations on a **void *** pointer are comparing **void *** pointers with other pointers, assigning addresses to **void *** pointers and casting **void *** pointers to valid pointer types.
- Pointers can be compared using the equality and relational operators. Comparisons using relational operators are meaningful only if the pointers point to members of the same array.
- Pointers that point to arrays can be subscripted exactly as array names can.
- An array name is equivalent to a constant pointer to the first element of the array.
- In pointer/offset notation, if the pointer points to the first element of the array, the offset is the same as an array subscript.
- All subscripted array expressions can be written with a pointer and an offset, using either the name of the array as a pointer or using a separate pointer that points to the array.
- Arrays may contain pointers.
- A pointer to a function is the address where the code for the function resides.

- Pointers to functions can be passed to functions, returned from functions, stored in arrays and assigned to other pointers.
- A common use of function pointers is in so-called menu-driven systems. The function pointers are used to select which function to call for a particular menu item.
- Function **strcpy** copies its second argument—a string—into its first argument—a character array. The programmer must ensure that the target array is large enough to store the string and its terminating null character.
- Function **strncpy** is equivalent to **strcpy**, except that a call to **strncpy** specifies the number of characters to be copied from the string into the array. The terminating null character will be copied only if the number of characters to be copied is at least one more than the length of the string.
- Function **strcat** appends its second string argument—including the terminating null character—to its first string argument. The first character of the second string replaces the null (**'\0'**) character of the first string. The programmer must ensure that the target array used to store the first string is large enough to store both the first string and the second string.
- Function **strncat** is equivalent to **strcat**, except that a call to **strncat** appends a specified number of characters from the second string to the first string. A terminating null character is appended to the result.
- Function **strcmp** compares its first string argument with its second string argument character by character. The function returns zero if the strings are equal, a negative value if the first string is less than the second string and a positive value if the first string is greater than the second string.
- Function **strncmp** is equivalent to **strcmp**, except that **strncmp** compares a specified number of characters. If the number of characters in one of the strings is less than the number of characters specified, **strncmp** compares characters until the null character in the shorter string is encountered.
- A sequence of calls to **strtok** breaks a string into tokens that are separated by characters contained in a second string argument. The first call specifies the string to be tokenized as the first argument, and subsequent calls to continue tokenizing the same string specify **NULL** as the first argument. The function returns a pointer to the current token from each call. If there are no more tokens when **strtok** is called, **NULL** is returned.
- Function **strlen** takes a string as an argument and returns the number of characters in the string—the terminating null character is not included in the length of the string.

TERMINOLOGY

add a pointer and an integer
address operator (**&**)
appending strings to other strings
array of pointers
array of strings
ASCII
<cctype>
character code
character constant
character pointer
character set
comparing strings
const
constant pointer
constant pointer to constant data
constant pointer to nonconstant data
copying strings
<cstring>
decrement a pointer
delimiter
dereference a pointer
dereferencing operator (*****)
directly reference a variable
EBCDIC
function pointer
increment a pointer
indefinite postponement
indirection
indirection operator (*****)
indirectly reference a variable
initialize a pointer
islower

length of a string
literal
nonconstant pointer to constant data
nonconstant pointer to nonconstant data
`NULL` pointer
numeric code of a character
offset
pass-by-reference
pass-by-value
pointer
pointer arithmetic
pointer assignment
pointer comparison
pointer expression
pointer/offset notation
pointer subscripting
pointer to a function
pointer to **`void`** (**`void *`**)
pointer types
principle of least privilege
`sizeof`
`strcat`
`strcmp`
`strcpy`
string
string concatenation
string constant
string literal
string processing
`strlen`
`strncat`
`strncmp`
`strncpy`
`strtok`
subtracting an integer from a pointer
subtracting two pointers
token
tokenizing strings
`toupper`
`void *` (pointer to **`void`**)
word processing

Terminology for Optional "Thinking About Objects" Section

collaboration
collaboration diagram
interaction
message
numbers in UML collaboration diagram
objects that participate in interaction
rectangle symbol in UML collaboration diagram
sequence of messages
solid line with arrowhead symbol in UML collaboration diagram
when interactions occur

SELF-REVIEW EXERCISES

5.1 Answer each of the following:

a) A pointer is a variable that contains as its value the _____ of another variable.
b) The three values that can be used to initialize a pointer are _____, _____ and _____.
c) The only integer that can be assigned directly to a pointer is _____.

5.2 State whether the following are *true* or *false*. If the answer is *false*, explain why.

a) The address operator **`&`** can be applied only to constants and to expressions.
b) A pointer that is declared to be of type **`void`** can be dereferenced.
c) Pointers of different types may not be assigned to one another without a cast operation.

5.3 For each of the following, write C++ statements that perform the specified task. Assume that double-precision, floating-point numbers are stored in eight bytes and that the starting address of the array is at location 1002500 in memory. Each part of the exercise should use the results of previous parts where appropriate.

a) Declare an array of type **`double`** called **`numbers`** with 10 elements, and initialize the elements to the values **`0.0`**, **`1.1`**, **`2.2`**, …, **`9.9`**. Assume that the symbolic constant **`SIZE`** has been defined as **`10`**.
b) Declare a pointer **`nPtr`** that points to a variable of type **`double`**.
c) Use a **`for`** structure to print the elements of array **`numbers`** using array subscript notation. Print each number with one position of precision to the right of the decimal point.

d) Write two separate statements that each assign the starting address of array **numbers** to the pointer variable **nPtr**.
e) Use a **for** structure to print the elements of array **numbers** using pointer/offset notation with pointer **nPtr**.
f) Use a **for** structure to print the elements of array **numbers** using pointer/offset notation with the array name as the pointer.
g) Use a **for** structure to print the elements of array **numbers** using pointer/subscript notation with pointer **nPtr**.
h) Refer to the fourth element of array **numbers** using array subscript notation, pointer/offset notation with the array name as the pointer, pointer subscript notation with **nPtr** and pointer/offset notation with **nPtr**.
i) Assuming that **nPtr** points to the beginning of array **numbers**, what address is referenced by **nPtr + 8**? What value is stored at that location?
j) Assuming that **nPtr** points to **numbers[5]**, what address is referenced by **nPtr** after **nPtr -= 4** is executed? What is the value stored at that location?

5.4 For each of the following, write a single statement that performs the specified task. Assume that floating-point variables **number1** and **number2** have been declared and that **number1** has been initialized to **7.3**. Assume that variable **ptr** is of type **char *** . Assume that arrays **s1** and **s2** are each 100-element **char** arrays that are initialized with string literals.

a) Declare the variable **fPtr** to be a pointer to an object of type **double**.
b) Assign the address of variable **number1** to pointer variable **fPtr**.
c) Print the value of the object pointed to by **fPtr**.
d) Assign the value of the object pointed to by **fPtr** to variable **number2**.
e) Print the value of **number2**.
f) Print the address of **number1**.
g) Print the address stored in **fPtr**. Is the value printed the same as the address of **number1**?
h) Copy the string stored in array **s2** into array **s1**.
i) Compare the string in **s1** with the string in **s2**, and print the result.
j) Append the first 10 characters from the string in **s2** to the string in **s1**.
k) Determine the length of the string in **s1**, and print the result.
l) Assign to **ptr** the location of the first token in **s2**. The tokens delimiters are commas (**,**).

5.5 Perform the task specified by each of the following statements:

a) Write the function header for a function called **exchange** that takes two pointers to double-precision, floating-point numbers **x** and **y** as parameters and does not return a value.
b) Write the function prototype for the function in part (a).
c) Write the function header for a function called **evaluate** that returns an integer and that takes as parameters integer **x** and a pointer to function **poly**. Function **poly** takes an integer parameter and returns an integer.
d) Write the function prototype for the function in part (c).
e) Write two statements that each initialize character array **vowel** with the string of vowels, **"AEIOU"**.

5.6 Find the error in each of the following program segments. Assume the following declarations and statements:

```
int *zPtr;        // zPtr will reference array z
int *aPtr = 0;
void *sPtr = 0;
int number;
int z[ 5 ] = { 1, 2, 3, 4, 5 };

sPtr = z;
```

a) `++zPtr;`
b)
```
// use pointer to get first value of array
number = zPtr;
```
c)
```
// assign array element 2 (the value 3) to number
number = *zPtr[ 2 ];
```
d)
```
// print entire array z
for ( int i = 0; i <= 5; i++ )
   cout << zPtr[ i ] << endl;
```
e)
```
// assign the value pointed to by sPtr to number
number = *sPtr;
```
f) `++z;`
g)
```
char s[ 10 ];
cout << strncpy( s, "hello", 5 ) << endl;
```
h)
```
char s[ 12 ];
strcpy( s, "Welcome Home" );
```
i)
```
if ( strcmp( string1, string2 ) )
   cout << "The strings are equal" << endl;
```

5.7 What (if anything) prints when each of the following statements is performed? If the statement contains an error, describe the error and indicate how to correct it. Assume the following variable declarations:

```
char s1[ 50 ] = "jack";
char s2[ 50 ] = "jill";
char s3[ 50 ];
```

a) `cout << strcpy( s3, s2 ) << endl;`
b)
```
cout << strcat( strcpy( s3, s1 ), " and " ), s2 )
     << endl;
```
c) `cout << strlen( s1 ) + strlen( s2 ) << endl;`
d) `cout << strlen( s3 ) << endl;`

ANSWERS TO SELF-REVIEW EXERCISES

5.1 a) address. b) **0**, **NULL**, an address. c) **0**.

5.2 a) False. 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.
b) False. A pointer to **void** cannot be dereferenced. Such a pointer does not have a type that enables the compiler to determine the number of bytes of memory to dereference.
c) False. Pointers of any type can be assigned to **void** pointers. Pointers of type **void** can be assigned to pointers of other types only with an explicit type cast.

5.3 a)
```
double numbers[ SIZE ] =
   { 0.0, 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9 };
```
b) `double *nPtr;`
c)
```
cout << fixed << showpoint << setprecision( 1 );
for ( int i = 0; i < SIZE; i++ )
   cout << numbers[ i ] << ' ';
```
d)
```
nPtr = numbers;
nPtr = &numbers[ 0 ];
```
e)
```
cout << fixed << showpoint << setprecision( 1 );
for ( int j = 0; j < SIZE; j++ )
   cout << *( nPtr + j ) << ' ';
```

f)
```
cout << fixed << showpoint << setprecision( 1 );
for ( int k = 0; k < SIZE; k++ )
   cout << *( numbers + k ) << ' ';
```
g)
```
cout << fixed << showpoint << setprecision( 1 );
for ( int m = 0; m < SIZE; m++ )
   cout << nPtr[ m ] << ' ';
```
h)
```
numbers[ 3 ]
*( numbers + 3 )
nPtr[ 3 ]
*( nPtr + 3 )
```
i) The address is **1002500 + 8 * 8 = 1002564**. The value is **8.8**.

j) The address of **numbers[5]** is **1002500 + 5 * 8 = 1002540**.
The address of **nPtr -= 4** is **1002540 - 4 * 8 = 1002508**.
The value at that location is **1.1**.

5.4

a)
```
double *fPtr;
```
b)
```
fPtr = &number1;
```
c)
```
cout << "The value of *fPtr is " << *fPtr << endl;
```
d)
```
number2 = *fPtr;
```
e)
```
cout << "The value of number2 is " << number2 << endl;
```
f)
```
cout << "The address of number1 is " << &number1 << endl;
```
g)
```
cout << "The address stored in fPtr is " << fPtr << endl;
```
Yes, the value is the same.

h)
```
strcpy( s1, s2 );
```
i)
```
cout << "strcmp(s1, s2) = " << strcmp( s1, s2 ) << endl;
```
j)
```
strncat( s1, s2, 10 );
```
k)
```
cout << "strlen(s1) = " << strlen( s1 ) << endl;
```
l)
```
ptr = strtok( s2, "," );
```

5.5

a)
```
void exchange( double *x, double *y )
```
b)
```
void exchange( double *, double * );
```
c)
```
int evaluate( int x, int (*poly)( int ) )
```
d)
```
int evaluate( int, int (*)( int ) );
```
e)
```
char vowel[] = "AEIOU";
char vowel[] = { 'A', 'E', 'I', 'O', 'U', '\0' };
```

5.6

a) Error: **zPtr** has not been initialized.
Correction: Initialize **zPtr** with **zPtr = z;**

b) Error: The pointer is not dereferenced.
Correction: Change the statement to **number = *zPtr;**

c) Error: **zPtr[2]** is not a pointer and should not be dereferenced.
Correction: Change ***zPtr[2]** to **zPtr[2]**.

d) Error: Referring to an array element outside the array bounds with pointer subscripting.
Correction: Change the relational operator in the **for** structure to **<** to prevent walking off the end of the array.

e) Error: Dereferencing a **void** pointer.
Correction: To dereference the **void** pointer, it must first be cast to an integer pointer. Change the preceding statement to **number = *((int *) sPtr);**

f) Error: Trying to modify an array name with pointer arithmetic.
Correction: Use a pointer variable instead of the array name to accomplish pointer arithmetic, or subscript the array name to refer to a specific element.

g) Error: Function **strncpy** does not write a terminating null character to array **s**, because its third argument is equal to the length of the string **"hello"**.

Correction: Make **6** the third argument of **strncpy** or assign **'\0'** to **s[5]** to ensure that the terminating null character is added to the string.

h) Error: Character array **s** is not large enough to store the terminating null character.
Correction: Declare the array with more elements.

i) Error: Function **strcmp** will return 0 if the strings are equal; therefore, the condition in the **if** structure will be false, and the output statement will not be executed.
Correction: Explicitly compare the result of **strcmp** with **0** in the condition of the **if** structure.

5.7 a) **jill**
b) **jack and jill**
c) **8**
d) **13**

EXERCISES

5.8 State whether the following are *true* or *false*. If *false*, explain why.

a) Two pointers that point to different arrays cannot be compared meaningfully.

b) Because the name of an array is a pointer to the first element of the array, array names can be manipulated in precisely the same manner as pointers.

5.9 For each of the following, write C++ statements that perform the specified task. Assume that unsigned integers are stored in two bytes and that the starting address of the array is at location 1002500 in memory.

a) Declare an array of type **unsigned int** called **values** with five elements, and initialize the elements to the even integers from 2 to 10. Assume that the symbolic constant **SIZE** has been defined as **5**.

b) Declare a pointer **vPtr** that points to an object of type **unsigned int**.

c) Use a **for** structure to print the elements of array **values** using array subscript notation.

d) Write two separate statements that assign the starting address of array **values** to pointer variable **vPtr**.

e) Use a **for** structure to print the elements of array **values** using pointer/offset notation.

f) Use a **for** structure to print the elements of array **values** using pointer/offset notation with the array name as the pointer.

g) Use a **for** structure to print the elements of array **values** by subscripting the pointer to the array.

h) Refer to the fifth element of **values** using array subscript notation pointer/offset notation with the array name as the pointer, pointer subscript notation and pointer/offset notation.

i) What address is referenced by **vPtr + 3**? What value is stored at that location?

j) Assuming that **vPtr** points to **values[4]**, what address is referenced by **vPtr -= 4**? What value is stored at that location?

5.10 For each of the following, write a single statement that performs the specified task. Assume that **long** integer variables **value1** and **value2** have been declared and that **value1** has been initialized to **200000**.

a) Declare the variable **longPtr** to be a pointer to an object of type **long**.

b) Assign the address of variable **value1** to pointer variable **longPtr**.

c) Print the value of the object pointed to by **longPtr**.

d) Assign the value of the object pointed to by **longPtr** to variable **value2**.

e) Print the value of **value2**.

f) Print the address of **value1**.

g) Print the address stored in **longPtr**. Is the value printed the same as **value1**'s address?

5.11 Perform the task specified by each of the following statements:

a) Write the function header for function **`zero`** that takes a long integer array parameter **`bigIntegers`** and does not return a value.

b) Write the function prototype for the function in part (a).

c) Write the function header for function **`add1AndSum`** that takes an integer array parameter **`oneTooSmall`** and returns an integer.

d) Write the function prototype for the function described in part (c).

Note: Exercise 5.12 through Exercise 5.15 are reasonably challenging. Once you have solved these problems, you ought to be able to implement most popular card games.

5.12 Modify the program in Fig. 5.24 so that the card dealing function deals a five-card poker hand. Then write functions to accomplish each of the following:

a) Determine whether the hand contains a pair.

b) Determine whether the hand contains two pairs.

c) Determine whether the hand contains three of a kind (e.g., three jacks).

d) Determine whether the hand contains four of a kind (e.g., four aces).

e) Determine whether the hand contains a flush (i.e., all five cards of the same suit).

f) Determine whether the hand contains a straight (i.e., five cards of consecutive face values).

5.13 Use the functions developed in Exercise 5.12 to write a program that deals two five-card poker hands, evaluates each hand and determines which is the better hand.

5.14 Modify the program developed in Exercise 5.13 so that it can simulate the dealer. The dealer's five-card hand is dealt "face down" so the player cannot see it. The program should then evaluate the dealer's hand, and, based on the quality of the hand, the dealer should draw one, two or three more cards to replace the corresponding number of unneeded cards in the original hand. The program should then reevaluate the dealer's hand. [*Caution:* This is a difficult problem!]

5.15 Modify the program developed in Exercise 5.14 so that it handles the dealer's hand, but the player is allowed to decide which cards of the player's hand to replace. The program should then evaluate both hands and determine who wins. Now use this new program to play 20 games against the computer. Who wins more games, you or the computer? Have one of your friends play 20 games against the computer. Who wins more games? Based on the results of these games, make appropriate modifications to refine your poker-playing program. [*Note:* This, too, is a difficult problem.] Play 20 more games. Does your modified program play a better game?

5.16 In the card-shuffling and dealing program of Fig. 5.24, we intentionally used an inefficient shuffling algorithm that introduced the possibility of indefinite postponement. In this problem, you will create a high-performance shuffling algorithm that avoids indefinite postponement.

Modify Fig. 5.24 as follows. Initialize the **`deck`** array as shown in Fig. 5.36. Modify the **`shuffle`** function to loop row-by-row and column-by-column through the array, touching every element once. Each element should be swapped with a randomly selected element of the array. Print the resulting array to determine whether the deck is satisfactorily shuffled (as in Fig. 5.37, for example). You may want your program to call the **`shuffle`** function several times to ensure a satisfactory shuffle.

Note that although the approach in this problem improves the shuffling algorithm, the dealing algorithm still requires searching the **`deck`** array for card 1, then card 2, then card 3 and so on. Worse yet, even after the dealing algorithm locates and deals the card, the algorithm continues searching through the remainder of the deck. Modify the program of Fig. 5.24 so that once a card is dealt, no further attempts are made to match that card number, and the program immediately proceeds with dealing the next card.

Unshuffled `deck` array

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	14	15	16	17	18	19	20	21	22	23	24	25	26
2	27	28	29	30	31	32	33	34	35	36	37	38	39
3	40	41	42	43	44	45	46	47	48	49	50	51	52

Fig. 5.36 Unshuffled **`deck`** array.

Sample shuffled `deck` array

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	19	40	27	25	36	46	10	34	35	41	18	2	44
1	13	28	14	16	21	30	8	11	31	17	24	7	1
2	12	33	15	42	43	23	45	3	29	32	4	47	26
3	50	38	52	39	48	51	9	5	37	49	22	6	20

Fig. 5.37 Sample shuffled **`deck`** array.

5.17 (*Simulation: The Tortoise and the Hare*) In this exercise, you will re-create the classic race of the tortoise and the hare. You will use random-number generation to develop a simulation of this memorable event.

Our contenders begin the race at "square 1" of 70 squares. Each square represents a possible position along the race course. The finish line is at square 70. The first contender to reach or pass square 70 is rewarded with a pail of fresh carrots and lettuce. The course weaves its way up the side of a slippery mountain, so occasionally the contenders lose ground.

There is a clock that ticks once per second. With each tick of the clock, your program should adjust the position of the animals according to the rules in Fig. 5.38.

Animal	Move type	Percentage of the time	Actual move
Tortoise	Fast plod	50%	3 squares to the right
	Slip	20%	6 squares to the left
	Slow plod	30%	1 square to the right
Hare	Sleep	20%	No move at all
	Big hop	20%	9 squares to the right
	Big slip	10%	12 squares to the left
	Small hop	30%	1 square to the right
	Small slip	20%	2 squares to the left

Fig. 5.38 Rules for moving the tortoise and the hare.

Use variables to keep track of the positions of the animals (i.e., position numbers are 1–70). Start each animal at position 1 (i.e., the "starting gate"). If an animal slips left before square 1, move the animal back to square 1.

Generate the percentages in the preceding table by producing a random integer i in the range $1 \le i \le 10$. For the tortoise, perform a "fast plod" when $1 \le i \le 5$, a "slip" when $6 \le i \le 7$ or a "slow plod" when $8 \le i \le 10$. Use a similar technique to move the hare.

Begin the race by printing

```
BANG !!!!!
AND THEY'RE OFF !!!!!
```

For each tick of the clock (i.e., each repetition of a loop), print a 70-position line showing the letter **T** in the tortoise's position and the letter **H** in the hare's position. Occasionally, the contenders land on the same square. In this case, the tortoise bites the hare and your program should print **OUCH!!!** beginning at that position. All print positions other than the **T**, the **H** or the **OUCH!!!** (in case of a tie) should be blank.

After printing each line, test if either animal has reached or passed square 70. If so, print the winner and terminate the simulation. If the tortoise wins, print **TORTOISE WINS!!! YAY!!!** If the hare wins, print **Hare wins. Yuch.** If both animals win on the same clock tick, you may want to favor the turtle (the "underdog"), or you may want to print **It's a tie**. If neither animal wins, perform the loop again to simulate the next tick of the clock. When you are ready to run your program, assemble a group of fans to watch the race. You'll be amazed how involved the audience gets!

SPECIAL SECTION: BUILDING YOUR OWN COMPUTER

In the next several problems, we take a temporary diversion away from the world of high-level-language programming. We "peel open" a computer and look at its internal structure. We introduce machine-language programming and write several machine-language programs. To make this an especially valuable experience, we then build a computer (using software-based *simulation*) on which you can execute your machine-language programs!

5.18 (*Machine-Language Programming*) Let us create a computer we will call the Simpletron. As its name implies, it is a simple machine, but, as we will soon see, a powerful one as well. The Simpletron runs programs written in the only language it directly understands, that is, Simpletron Machine Language, or SML for short.

The Simpletron contains an *accumulator*—a "special register" in which information is put before the Simpletron uses that information in calculations or examines it in various ways. All information in the Simpletron is handled in terms of *words*. A word is a signed four-digit decimal number, such as **+3364**, **-1293**, **+0007**, **-0001**, etc. The Simpletron is equipped with a 100-word memory and these words are referenced by their location numbers **00**, **01**, …, **99**.

Before running an SML program, we must *load,* or place, the program into memory. The first instruction (or statement) of every SML program is always placed in location **00**. The simulator will start executing at this location.

Each instruction written in SML occupies one word of the Simpletron's memory; thus, instructions are signed four-digit decimal numbers. Assume that the sign of an SML instruction is always plus, but the sign of a data word may be either plus or minus. Each location in the Simpletron's memory may contain an instruction, a data value used by a program or an unused (and hence undefined) area of memory. The first two digits of each SML instruction are the *operation code* that specifies the operation to be performed. SML operation codes are shown in Fig. 5.39.

The last two digits of an SML instruction are the *operand*—the address of the memory location containing the word to which the operation applies.

Operation code	Meaning
Input/output operations:	
`const int READ = 10;`	Read a word from the keyboard into a specific location in memory.
`const int WRITE = 11;`	Write a word from a specific location in memory to the screen.
Load and store operations:	
`const int LOAD = 20;`	Load a word from a specific location in memory into the accumulator.
`const int STORE = 21;`	Store a word from the accumulator into a specific location in memory.
Arithmetic operations:	
`const int ADD = 30;`	Add a word from a specific location in memory to the word in the accumulator (leave result in accumulator).
`const int SUBTRACT = 31;`	Subtract a word from a specific location in memory from the word in the accumulator (leave result in accumulator).
`const int DIVIDE = 32;`	Divide a word from a specific location in memory into the word in the accumulator (leave result in accumulator).
`const int MULTIPLY = 33;`	Multiply a word from a specific location in memory by the word in the accumulator (leave result in accumulator).
Transfer-of-control operations:	
`const int BRANCH = 40;`	Branch to a specific location in memory.
`const int BRANCHNEG = 41;`	Branch to a specific location in memory if the accumulator is negative.
`const int BRANCHZERO = 42;`	Branch to a specific location in memory if the accumulator is zero.
`const int HALT = 43;`	Halt—the program has completed its task.

Fig. 5.39 Simpletron Machine Language (SML) operation codes.

Now let us consider two simple SML programs. The first SML program (Fig. 5.40) reads two numbers from the keyboard and computes and prints their sum. The instruction **`+1007`** reads the first number from the keyboard and places it into location **`07`** (which has been initialized to zero). Instruction **`+1008`** reads the next number into location **`08`**. The *load* instruction, **`+2007`**, places (copies) the first number into the accumulator, and the *add* instruction, **`+3008`**, adds the second number to the number in the accumulator. *All SML arithmetic instructions leave their results in the accumulator.* The *store* instruction, **`+2109`**, places (copies) the result back into memory location **`09`**. Then the *write* instruction, **`+1109`**, takes the number and prints it (as a signed four-digit decimal number). The *halt* instruction, **`+4300`**, terminates execution.

Location	Number	Instruction
00	+1007	(Read A)
01	+1008	(Read B)
02	+2007	(Load A)
03	+3008	(Add B)
04	+2109	(Store C)
05	+1109	(Write C)
06	+4300	(Halt)
07	+0000	(Variable A)
08	+0000	(Variable B)
09	+0000	(Result C)

Fig. 5.40 SML Example 1.

The SML program in Fig. 5.41 reads two numbers from the keyboard, then determines and prints the larger value. Note the use of the instruction **+4107** as a conditional transfer of control, much the same as C++'s **if** statement.

Now write SML programs to accomplish each of the following tasks:

a) Use a sentinel-controlled loop to read positive numbers and compute and print their sum. Terminate input when a negative number is entered.

b) Use a counter-controlled loop to read seven numbers, some positive and some negative, and compute and print their average.

c) Read a series of numbers, and determine and print the largest number. The first number read indicates how many numbers should be processed.

Location	Number	Instruction
00	+1009	(Read A)
01	+1010	(Read B)
02	+2009	(Load A)
03	+3110	(Subtract B)
04	+4107	(Branch negative to 07)
05	+1109	(Write A)
06	+4300	(Halt)
07	+1110	(Write B)
08	+4300	(Halt)
09	+0000	(Variable A)
10	+0000	(Variable B)

Fig. 5.41 SML Example 2.

5.19 (*Computer Simulator*) It may at first seem outrageous, but in this problem, you are going to build your own computer. No, you will not be soldering components together. Rather, you will use the powerful technique of *software-based simulation* to create a *software model* of the Simpletron. You will not be disappointed. Your Simpletron simulator will turn the computer you are using into a Simpletron, and you actually will be able to run, test and debug the SML programs you wrote in Exercise 5.18.

When you run your Simpletron simulator, it should begin by printing

```
*** Welcome to Simpletron! ***

*** Please enter your program one instruction ***
*** (or data word) at a time. I will type the ***
*** location number and a question mark (?).  ***
*** You then type the word for that location. ***
*** Type the sentinel -99999 to stop entering ***
*** your program. ***
```

Your program should simulate the Simpletron's memory with a single-subscripted, 100-element array **memory**. Now assume that the simulator is running, and let us examine the dialog as we enter the program of Example 2 of Exercise 5.18:

```
00 ? +1009
01 ? +1010
02 ? +2009
03 ? +3110
04 ? +4107
05 ? +1109
06 ? +4300
07 ? +1110
08 ? +4300
09 ? +0000
10 ? +0000
11 ? -99999

*** Program loading completed ***
*** Program execution begins  ***
```

Note that the numbers to the right of each **?** in the preceding dialog represent the SML program instructions input by the user.

The SML program has now been placed (or loaded) into array **memory**. Now the Simpletron executes your SML program. Execution begins with the instruction in location **00** and, like C++, continues sequentially, unless directed to some other part of the program by a transfer of control.

Use variable **accumulator** to represent the accumulator register. Use variable **counter** to keep track of the location in memory that contains the instruction being performed. Use variable **operationCode** to indicate the operation currently being performed (i.e., the left two digits of the instruction word). Use variable **operand** to indicate the memory location on which the current instruction operates. Thus, **operand** is the rightmost two digits of the instruction currently being performed. Do not execute instructions directly from memory. Rather, transfer the next instruction to be performed from memory to a variable called **instructionRegister**. Then "pick off" the left two digits and place them in **operationCode**, and "pick off" the right two digits and place them in **operand**. When Simpletron begins execution, the special registers are all initialized to zero.

Now let us "walk through" the execution of the first SML instruction, **+1009** in memory location **00**. This is called an *instruction execution cycle.*

The **counter** tells us the location of the next instruction to be performed. We *fetch* the contents of that location from **memory** by using the C++ statement

```
instructionRegister = memory[ counter ];
```

The operation code and operand are extracted from the instruction register by the statements

```
operationCode = instructionRegister / 100;
operand = instructionRegister % 100;
```

Now, the Simpletron must determine that the operation code is actually a *read* (versus a *write*, a *load*, etc.). A **switch** differentiates among the 12 operations of SML.

In the **switch** structure, the behavior of various SML instructions is simulated as follows (we leave the others to the reader):

read:	`cin >> memory[ operand ];`
load:	`accumulator = memory[ operand ];`
add:	`accumulator += memory[ operand ];`
branch:	We will discuss the branch instructions shortly.
halt:	This instruction prints the message `*** Simpletron execution terminated ***`

The *halt* instruction also causes the Simpletron to print the name and contents of each register, as well as the complete contents of memory. Such a printout is often called a *computer dump* (and, no, a computer dump is not a place where old computers go). To help you program your dump function, a sample dump format is shown in Fig. 5.42. Note that a dump after executing a Simpletron program would show the actual values of instructions and data values at the moment execution terminated. To format numbers with their sign as shown in the dump, use stream manipulator ***showpos***. To disable the display of the sign use stream manipulator ***noshowpos***. For numbers that have fewer than four digits, you can format numbers with leading zeros between the sign and the value by using the following statement before outputting the value:

```
cout << setfill( '0' ) << internal;
```

Parameterized stream manipulator ***setfill*** (from header **<iomanip>**) specifies the fill character that will appear between the sign and the value when a number is displayed with a field width of five characters, but does not have four digits. (One position in the field width is reserved for the sign.) Stream manipulator ***internal*** indicates that the fill characters should appear between the sign and the numeric value.

Let us proceed with the execution of our program's first instruction—**+1009** in location **00**. As we have indicated, the **switch** structure simulates this by performing the C++ statement

```
cin >> memory[ operand ];
```

A question mark (**?**) should be displayed on the screen before the **cin** statement executes to prompt the user for input. The Simpletron waits for the user to type a value and press the *Enter* key. The value is then read into location **09**.

At this point, simulation of the first instruction is complete. All that remains is to prepare the Simpletron to execute the next instruction. The instruction just performed was not a transfer of control, so we need merely increment the instruction counter register as follows:

```
++counter;
```

This completes the simulated execution of the first instruction. The entire process (i.e., the instruction execution cycle) begins anew with the fetch of the next instruction to execute.

```
REGISTERS:
accumulator          +0000
counter                 00
instructionRegister  +0000
operationCode           00
operand                 00

MEMORY:
       0     1     2     3     4     5     6     7     8     9
 0 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
10 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
20 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
30 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
40 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
50 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
60 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
70 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
80 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
90 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
```

Fig. 5.42 A sample dump.

Now let us consider how to simulate the branching instructions (i.e., the transfers of control). All we need to do is adjust the value in the instruction counter appropriately. Therefore, the unconditional branch instruction (**40**) is simulated in the **switch** as

```
counter = operand;
```

The conditional "branch if accumulator is zero" instruction is simulated as

```
if ( accumulator == 0 )
   counter = operand;
```

At this point, you should implement your Simpletron simulator and run each of the SML programs you wrote in Exercise 5.18. You may embellish SML with additional features and provide for these in your simulator.

Your simulator should check for various types of errors. During the program loading phase, for example, each number the user types into the Simpletron's **memory** must be in the range **-9999** to **+9999**. Your simulator should use a **while** loop to test that each number entered is in this range and, if not, keep prompting the user to reenter the number until the user enters a correct number.

During the execution phase, your simulator should check for various serious errors, such as attempts to divide by zero, attempts to execute invalid operation codes, accumulator overflows (i.e., arithmetic operations resulting in values larger than **+9999** or smaller than **-9999**) and the like. Such serious errors are called *fatal errors*. When a fatal error is detected, your simulator should print an error message such as

```
*** Attempt to divide by zero ***
*** Simpletron execution abnormally terminated ***
```

and should print a full computer dump in the format we have discussed previously. This will help the user locate the error in the program.

MORE POINTER EXERCISES

5.20 Modify the card-shuffling and dealing program of Fig. 5.24 so the shuffling and dealing operations are performed by the same function (**`shuffleAndDeal`**). The function should contain one nested looping structure that is similar to function **`shuffle`** in Fig. 5.24.

5.21 What does this program do?

```
// Ex. 5.21: ex05_21.cpp
// What does this program do?
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

void mystery1( char *, const char * );  // prototype

int main()
{
   char string1[ 80 ];
   char string2[ 80 ];

   cout << "Enter two strings: ";
   cin >> string1 >> string2;
   mystery1( string1, string2 );
   cout << string1 << endl;

   return 0;  // indicates successful termination

} // end main

// What does this function do?
void mystery1( char *s1, const char *s2 )
{
   while ( *s1 != '\0' )
      ++s1;

   for ( ; *s1 = *s2; s1++, s2++ )
      ;   // empty statement

} // end function mystery1
```

5.22 What does this program do?

```
// Ex. 5.22: ex05_22.cpp
// What does this program do?
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

int mystery2( const char * );  // prototype
```

```
10
11 int main()
12 {
13    char string1[ 80 ];
14
15    cout << "Enter a string: ";
16    cin >> string1;
17    cout << mystery2( string1 ) << endl;
18
19    return 0;  // indicates successful termination
20
21 } // end main
22
23 // What does this function do?
24 int mystery2( const char *s )
25 {
26    int x;
27
28    for ( x = 0; *s != '\0'; s++ )
29       ++x;
30
31    return x;
32
33 } // end function mystery2
```

5.23 Find the error in each of the following segments. If the error can be corrected, explain how.

a)
```
int *number;
cout << number << endl;
```

b)
```
double *realPtr;
long *integerPtr;
integerPtr = realPtr;
```

c)
```
int * x, y;
x = y;
```

d)
```
char s[] = "this is a character array";
for ( ; *s != '\0'; s++)
   cout << *s << ' ';
```

e)
```
short *numPtr, result;
void *genericPtr = numPtr;
result = *genericPtr + 7;
```

f)
```
double x = 19.34;
double xPtr = &x;
cout << xPtr << endl;
```

g)
```
char *s;
cout << s << endl;
```

5.24 (*Quicksort*) In the examples and exercises of Chapter 4, we discussed the sorting techniques of the bubble sort, bucket sort and selection sort. We now present the recursive sorting technique called Quicksort. The basic algorithm for a single-subscripted array of values is as follows:

a) *Partitioning Step:* Take the first element of the unsorted array and determine its final location in the sorted array (i.e., all values to the left of the element in the array are less than the element, and all values to the right of the element in the array are greater than the element). We now have one element in its proper location and two unsorted subarrays.

b) *Recursive Step:* Perform step 1 on each unsorted subarray.

Each time step 1 is performed on a subarray, another element is placed in its final location of the sorted array, and two unsorted subarrays are created. When a subarray consists of one element, that subarray must be sorted; therefore, that element is in its final location.

The basic algorithm seems simple enough, but how do we determine the final position of the first element of each subarray? As an example, consider the following set of values (the element in bold is the partitioning element—it will be placed in its final location in the sorted array):

37 2 6 4 89 8 10 12 68 45

a) Starting from the rightmost element of the array, compare each element with **37** until an element less than **37** is found. Then swap **37** and that element. The first element less than **37** is 12, so **37** and 12 are swapped. The values now reside in the array as follows:

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

Element 12 is in italics to indicate that it was just swapped with **37**.

b) Starting from the left of the array, but beginning with the element after 12, compare each element with **37** until an element greater than **37** is found. Then swap **37** and that element. The first element greater than **37** is 89, so **37** and 89 are swapped. The values now reside in the array as follows:

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

c) Starting from the right, but beginning with the element before 89, compare each element with **37** until an element less than **37** is found. Then swap **37** and that element. The first element less than **37** is 10, so **37** and 10 are swapped. The values now reside in the array as follows:

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

d) Starting from the left, but beginning with the element after 10, compare each element with **37** until an element greater than **37** is found. Then swap **37** and that element. There are no more elements greater than **37**, so when we compare **37** with itself, we know that **37** has been placed in its final location of the sorted array.

Once the partition has been applied to the array, there are two unsorted subarrays. The subarray with values less than 37 contains 12, 2, 6, 4, 10 and 8. The subarray with values greater than 37 contains 89, 68 and 45. The sort continues with both subarrays being partitioned in the same manner as the original array.

Based on the preceding discussion, write recursive function **`quickSort`** to sort a single-subscripted integer array. The function should receive as arguments an integer array, a starting subscript and an ending subscript. Function **`partition`** should be called by **`quickSort`** to perform the partitioning step.

5.25 (*Maze Traversal*) The grid of hashes (**`#`**) and dots (**`.`**) in Fig. 5.43 is a double-subscripted array representation of a maze. In the double-subscripted array, the hashes (**`#`**) represent the walls of the maze and the dots represent squares in the possible paths through the maze. Moves can be made only to a location in the array that contains a dot.

There is a simple algorithm for walking through a maze that guarantees finding the exit (assuming that there is an exit). If there is not an exit, you will arrive at the starting location again. Place your right hand on the wall to your right and begin walking forward. Never remove your hand from the wall. If the maze turns to the right, you follow the wall to the right. As long as you do not remove your hand from the wall, eventually you will arrive at the exit of the maze. There may be a shorter path than the one you have taken, but you are guaranteed to get out of the maze if you follow the algorithm.

```
# # # # # # # # # # # #
# . . . # . . . . . . #
. . # . # . # # # # . #
# # # . # . . . . # . #
# . . . . # # # . # . .
# # # # . # . # . # . #
# . . # . # . # . # . #
# # . # . # . # . # . #
# . . . . . . . . # . #
# # # # # # . # # # . #
# . . . . . . # . . . #
# # # # # # # # # # # #
```

Fig. 5.43 Double-subscripted array representation of a maze.

Write recursive function **mazeTraverse** to walk through the maze. The function should receive as arguments a 12-by-12 character array representing the maze and the starting location of the maze. As **mazeTraverse** attempts to locate the exit from the maze, it should place the character **X** in each square in the path. The function should display the maze after each move so the user can watch as the maze is solved.

5.26 (*Generating Mazes Randomly*) Write a function **mazeGenerator** that takes as an argument a double-subscripted 12-by-12 character array and randomly produces a maze. The function should also provide the starting and ending locations of the maze. Try your function **mazeTraverse** from Exercise 5.25 using several randomly generated mazes.

5.27 (*Mazes of Any Size*) Generalize functions **mazeTraverse** and **mazeGenerator** of Exercise 5.25 and Exercise 5.26 to process mazes of any width and height.

5.28 (*Arrays of Pointers to Functions*) Rewrite the program of Fig. 4.23 to use a menu-driven interface. The program should offer the user five options as follows (these should be displayed on the screen):

```
Enter a choice:
  0  Print the array of grades
  1  Find the minimum grade
  2  Find the maximum grade
  3  Print the average on all tests for each student
  4  End program
```

One restriction on using arrays of pointers to functions is that all the pointers must have the same type. The pointers must be to functions of the same return type that receive arguments of the same type. For this reason, the functions in Fig. 4.23 must be modified so they each return the same type and take the same parameters. Modify functions **minimum** and **maximum** to print the minimum or maximum value and return nothing. For option 3, modify function **average** of Fig. 4.23 to output the average for each student (not a specific student). Function **average** should return nothing and take the same parameters as **printArray**, **minimum** and **maximum**. Store the pointers to the four functions in array **processGrades**, and use the choice made by the user as the subscript into the array for calling each function.

5.29 (*Modifications to the Simpletron Simulator*) In Exercise 5.19, you wrote a software simulation of a computer that executes programs written in Simpletron Machine Language (SML). In this

exercise, we propose several modifications and enhancements to the Simpletron Simulator. In Exercise 17.26 and Exercise 17.27, we propose building a compiler that converts programs written in a high-level programming language (a variation of BASIC) to SML. Some of the following modifications and enhancements may be required to execute the programs produced by the compiler. (*Note:* Some modifications may conflict with others and therefore must be done separately.)

a) Extend the Simpletron Simulator's memory to contain 1000 memory locations to enable the Simpletron to handle larger programs.
b) Allow the simulator to perform modulus calculations. This requires an additional Simpletron Machine Language instruction.
c) Allow the simulator to perform exponentiation calculations. This requires an additional Simpletron Machine Language instruction.
d) Modify the simulator to use hexadecimal values rather than integer values to represent Simpletron Machine Language instructions.
e) Modify the simulator to allow output of a newline. This requires an additional Simpletron Machine Language instruction.
f) Modify the simulator to process floating-point values in addition to integer values.
g) Modify the simulator to handle string input. [*Hint:* Each Simpletron word can be divided into two groups, each holding a two-digit integer. Each two-digit integer represents the ASCII decimal equivalent of a character. Add a machine-language instruction that will input a string and store the string beginning at a specific Simpletron memory location. The first half of the word at that location will be a count of the number of characters in the string (i.e., the length of the string). Each succeeding half-word contains one ASCII character expressed as two decimal digits. The machine-language instruction converts each character into its ASCII equivalent and assigns it to a half-word.]
h) Modify the simulator to handle output of strings stored in the format of part (g). [*Hint:* Add a machine-language instruction that will print a string beginning at a certain Simpletron memory location. The first half of the word at that location is a count of the number of characters in the string (i.e., the length of the string). Each succeeding half-word contains one ASCII character expressed as two decimal digits. The machine-language instruction checks the length and prints the string by translating each two-digit number into its equivalent character.]
i) Modify the simulator to include instruction **`SML_DEBUG`** that prints a memory dump after each instruction executes. Give **`SML_DEBUG`** an operation code of **`44`**. The word **`+4401`** turns on debug mode, and **`+4400`** turns off debug mode.

5.30 What does this program do?

```
1  // Ex. 5.30: ex05_30.cpp
2  // What does this program do?
3  #include <iostream>
4
5  using std::cout;
6  using std::cin;
7  using std::endl;
8
9  bool mystery3( const char *, const char * );  // prototype
10
11 int main()
12 {
13    char string1[ 80 ], string2[ 80 ];
14
```

```
   cout << "Enter two strings: ";
   cin >> string1 >> string2;
   cout << "The result is "
        << mystery3( string1, string2 ) << endl;

   return 0;  // indicates successful termination

} // end main

// What does this function do?
bool mystery3( const char *s1, const char *s2 )
{
   for ( ; *s1 != '\0' && *s2 != '\0'; s1++, s2++ )

      if ( *s1 != *s2 )
         return false;

   return true;

} // end function mystery3
```

STRING-MANIPULATION EXERCISES

5.31 Write a program that uses function **strcmp** to compare two strings input by the user. The program should state whether the first string is less than, equal to or greater than the second string.

5.32 Write a program that uses function **strncmp** to compare two strings input by the user. The program should input the number of characters to compare. The program should state whether the first string is less than, equal to or greater than the second string.

5.33 Write a program that uses random-number generation to create sentences. The program should use four arrays of pointers to **char** called **article**, **noun**, **verb** and **preposition**. The program should create a sentence by selecting a word at random from each array in the following order: **article**, **noun**, **verb**, **preposition**, **article** and **noun**. As each word is picked, it should be concatenated to the previous words in an array that is large enough to hold the entire sentence. The words should be separated by spaces. When the final sentence is output, it should start with a capital letter and end with a period. The program should generate 20 such sentences.

The arrays should be filled as follows: The **article** array should contain the articles **"the"**, **"a"**, **"one"**, **"some"** and **"any"**; the **noun** array should contain the nouns **"boy"**, **"girl"**, **"dog"**, **"town"** and **"car"**; the **verb** array should contain the verbs **"drove"**, **"jumped"**, **"ran"**, **"walked"** and **"skipped"**; the **preposition** array should contain the prepositions **"to"**, **"from"**, **"over"**, **"under"** and **"on"**.

After completing the program, modify it to produce a short story consisting of several of these sentences. (How about the possibility of a random term-paper writer!)

5.34 *(Limericks)* A limerick is a humorous five-line verse in which the first and second lines rhyme with the fifth, and the third line rhymes with the fourth. Using techniques similar to those developed in Exercise 5.33, write a C++ program that produces random limericks. Polishing this program to produce good limericks is a challenging problem, but the result will be worth the effort!

5.35 Write a program that encodes English language phrases into pig Latin. Pig Latin is a form of coded language often used for amusement. Many variations exist in the methods used to form pig Latin phrases. For simplicity, use the following algorithm: To form a pig-Latin phrase from an English-language phrase, tokenize the phrase into words with function **strtok**. To translate each English

word into a pig-Latin word, place the first letter of the English word at the end of the English word and add the letters "**ay**." Thus, the word "**jump**" becomes "**umpjay**," the word "**the**" becomes "**hetay**" and the word "**computer**" becomes "**omputercay**." Blanks between words remain as blanks. Assume that the English phrase consists of words separated by blanks, there are no punctuation marks and all words have two or more letters. Function **printLatinWord** should display each word. (*Hint:* Each time a token is found in a call to **strtok**, pass the token pointer to function **printLatinWord** and print the pig-Latin word.)

5.36 Write a program that inputs a telephone number as a string in the form **(555) 555-5555**. The program should use function **strtok** to extract the area code as a token, the first three digits of the phone number as a token, and the last four digits of the phone number as a token. The seven digits of the phone number should be concatenated into one string. Both the area code and the phone number should be printed.

5.37 Write a program that inputs a line of text, tokenizes the line with function **strtok** and outputs the tokens in reverse order.

5.38 Use the string comparison functions discussed in Section 5.12.2 and the techniques for sorting arrays developed in Chapter 4 to write a program that alphabetizes a list of strings. Use the names of 10 or 15 towns in your area as data for your program.

5.39 Write two versions of each string copy and string concatenation function in Fig. 5.27. The first version should use array subscripting, and the second should use pointers and pointer arithmetic.

5.40 Write two versions of each string comparison function in Fig. 5.27. The first version should use array subscripting, and the second version should use pointers and pointer arithmetic.

5.41 Write two versions of function **strlen** in Fig. 5.27. The first version should use array subscripting, and the second version should use pointers and pointer arithmetic.

SPECIAL SECTION: ADVANCED STRING-MANIPULATION EXERCISES

The preceding exercises are keyed to the text and designed to test the reader's understanding of fundamental string-manipulation concepts. This section includes a collection of intermediate and advanced string-manipulation exercises. The reader should find these problems challenging, yet enjoyable. The problems vary considerably in difficulty. Some require an hour or two of program writing and implementation. Others are useful for lab assignments that might require two or three weeks of study and implementation. Some are challenging term projects.

5.42 *(Text Analysis)* The availability of computers with string-manipulation capabilities has resulted in some rather interesting approaches to analyzing the writings of great authors. Much attention has been focused on whether William Shakespeare ever lived. Some scholars believe there is substantial evidence indicating that Christopher Marlowe or other authors actually penned the masterpieces attributed to Shakespeare. Researchers have used computers to find similarities in the writings of these two authors. This exercise examines three methods for analyzing texts with a computer.

a) Write a program that reads several lines of text from the keyboard and prints a table indicating the number of occurrences of each letter of the alphabet in the text. For example, the phrase

```
To be, or not to be: that is the question:
```

contains one "a," two "b's," no "c's," etc.

b) Write a program that reads several lines of text and prints a table indicating the number of one-letter words, two-letter words, three-letter words, etc., appearing in the text. For example, the phrase

```
Whether 'tis nobler in the mind to suffer
```

contains the following word lengths and occurrences:

Word length	Occurrences
1	0
2	2
3	1
4	2 (including `'tis`)
5	0
6	2
7	1

c) Write a program that reads several lines of text and prints a table indicating the number of occurrences of each different word in the text. The first version of your program should include the words in the table in the same order in which they appear in the text. For example, the lines

```
To be, or not to be: that is the question:
Whether 'tis nobler in the mind to suffer
```

contain the words "to" three times, the word "be" two times, the word "or" once, etc. A more interesting (and useful) printout should then be attempted in which the words are sorted alphabetically.

5.43 *(Word Processing)* One important function in word-processing systems is *type justification*—the alignment of words to both the left and right margins of a page. This generates a professional-looking document that gives the appearance of being set in type rather than prepared on a typewriter. Type justification can be accomplished on computer systems by inserting blank characters between each of the words in a line so that the rightmost word aligns with the right margin.

Write a program that reads several lines of text and prints this text in type-justified format. Assume that the text is to be printed on 8-1/2-inch-wide paper and that one-inch margins are to be allowed on both the left and right sides of the printed page. Assume that the computer prints 10 characters to the horizontal inch. Therefore, your program should print 6-1/2 inches of text, or 65 characters per line.

5.44 *(Printing Dates in Various Formats)* Dates are commonly printed in several different formats in business correspondence. Two of the more common formats are

```
07/21/1955
July 21, 1955
```

Write a program that reads a date in the first format and prints that date in the second format.

5.45 *(Check Protection)* Computers are frequently employed in check-writing systems such as payroll and accounts payable applications. Many strange stories circulate regarding weekly paychecks being printed (by mistake) for amounts in excess of $1 million. Weird amounts are printed by computerized check-writing systems, because of human error or machine failure. Systems designers build controls into their systems to prevent such erroneous checks from being issued.

Another serious problem is the intentional alteration of a check amount by someone who intends to cash a check fraudulently. To prevent a dollar amount from being altered, most computerized check-writing systems employ a technique called *check protection.*

Checks designed for imprinting by computer contain a fixed number of spaces in which the computer may print an amount. Suppose that a paycheck contains eight blank spaces in which the computer is supposed to print the amount of a weekly paycheck. If the amount is large, then all eight of those spaces will be filled, for example,

```
1,230.60    (check amount)
--------
12345678    (position numbers)
```

On the other hand, if the amount is less than $1000, then several of the spaces would ordinarily be left blank. For example,

```
   99.87
--------
12345678
```

contains three blank spaces. If a check is printed with blank spaces, it is easier for someone to alter the amount of the check. To prevent a check from being altered, many check-writing systems insert *leading asterisks* to protect the amount as follows:

```
***99.87
--------
12345678
```

Write a program that inputs a dollar amount to be printed on a check and then prints the amount in check-protected format with leading asterisks if necessary. Assume that nine spaces are available for printing an amount.

5.46 *(Writing the Word Equivalent of a Check Amount)* Continuing the discussion of the previous example, we reiterate the importance of designing check-writing systems to prevent alteration of check amounts. One common security method requires that the check amount be written both in numbers and "spelled out" in words. Even if someone is able to alter the numerical amount of the check, it is extremely difficult to change the amount in words.

Write a program that inputs a numeric check amount and writes the word equivalent of the amount. Your program should be able to handle check amounts as large as $99.99. For example, the amount 112.43 should be written as

```
ONE HUNDRED TWELVE and 43/100
```

5.47 *(Morse Code)* Perhaps the most famous of all coding schemes is the Morse code, developed by Samuel Morse in 1832 for use with the telegraph system. The Morse code assigns a series of dots and dashes to each letter of the alphabet, each digit and a few special characters (such as period, comma, colon and semicolon). In sound-oriented systems, the dot represents a short sound, and the dash represents a long sound. Other representations of dots and dashes are used with light-oriented systems and signal-flag systems.

Separation between words is indicated by a space, or, quite simply, the absence of a dot or dash. In a sound-oriented system, a space is indicated by a short period of time during which no sound is transmitted. The international version of the Morse code appears in Fig. 5.44.

Write a program that reads an English-language phrase and encodes the phrase into Morse code. Also write a program that reads a phrase in Morse code and converts the phrase into the English-language equivalent. Use one blank between each Morse-coded letter and three blanks between each Morse-coded word.

Character	Code	Character	Code
A	.-	T	-
B	-...	U	..-
C	-.-.	V	...-
D	-..	W	.--
E	.	X	-..-
F	..-.	Y	-.--
G	--.	Z	--..
H	....		
I	..	*Digits*	
J	.---	1	.----
K	-.-	2	..---
L	.-..	3	...--
M	--	4	....-
N	-.	5	.....
O	---	6	-....
P	.--.	7	--...
Q	--.-	8	---..
R	.-.	9	----.
S	...	0	-----

Fig. 5.44 Morse code alphabet.

5.48 *(A Metric Conversion Program)* Write a program that will assist the user with metric conversions. Your program should allow the user to specify the names of the units as strings (i.e., centimeters, liters, grams, etc., for the metric system and inches, quarts, pounds, etc., for the English system) and should respond to simple questions such as

```
"How many inches are in 2 meters?"
"How many liters are in 10 quarts?"
```

Your program should recognize invalid conversions. For example, the question

```
"How many feet in 5 kilograms?"
```

is not meaningful, because `"feet"` are units of length, while `"kilograms"` are units of weight.

A CHALLENGING STRING-MANIPULATION PROJECT

5.49 *(A Crossword Puzzle Generator)* Most people have worked a crossword puzzle, but few have ever attempted to generate one. Generating a crossword puzzle is a difficult problem. It is suggested here as a string-manipulation project requiring substantial sophistication and effort. There are many issues that the programmer must resolve to get even the simplest crossword puzzle generator program working. For example, how does one represent the grid of a crossword puzzle inside the computer?

Should one use a series of strings, or should double-subscripted arrays be used? The programmer needs a source of words (i.e., a computerized dictionary) that can be directly referenced by the program. In what form should these words be stored to facilitate the complex manipulations required by the program? The really ambitious reader will want to generate the "clues" portion of the puzzle in which the brief hints for each "across" word and each "down" word are printed for the puzzle worker. Merely printing a version of the blank puzzle itself is not a simple problem.