

# 7

---

## C Pointers

---

### Objectives

- To be able to use pointers.
- To be able to use pointers to pass arguments to functions using call 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

*Hamlet*

*Many things, having full reference*

*To one consent, may work contrariously.*

William Shakespeare

*King Henry V*

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

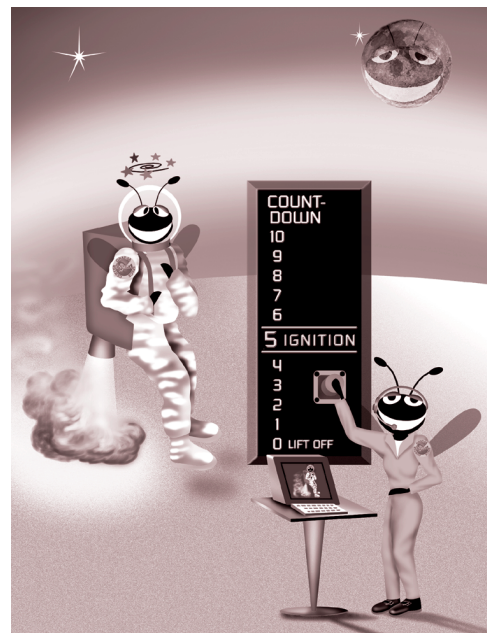
Dr. Routh

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

Ken Thompson

1983 Turing Award Lecture

Association for Computing Machinery, Inc.



## Outline

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

*Summary • Terminology • Common Programming Errors • Good Programming Practices • Performance Tips • Portability Tips • Self-Review Exercises • Answers to Self-Review Exercises • Exercises • Special Section: Building Your Own Computer*

## 7.1 Introduction

In this chapter, we discuss one of the most powerful features of the C programming language, the *pointer*. Pointers are among C's most difficult capabilities to master. Pointers enable programs to simulate call by reference, and to create and manipulate dynamic data structures, i.e., data structures that can grow and shrink, such as linked lists, queues, stacks and trees. This chapter explains basic pointer concepts. Chapter 10 examines the use of pointers with structures. Chapter 12 introduces dynamic memory management techniques and presents examples of creating and using dynamic data structures.

## 7.2 Pointer Variable Declarations and Initialization

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

Pointers, like any other variables, must be declared before they can be used. The declaration

```
int *countPtr, count;
```

declares the variable `countPtr` to be of type `int *` (i.e., a pointer to an integer value) and is read, “`countPtr` is a pointer to `int`” or “`countPtr` points to an object of type integer.” Also, the variable `count` is declared to be an integer, not a pointer to an integer. The `*` only applies to `countPtr` in the declaration. When `*` is used in this manner in a declaration, it indicates that the variable being declared is a pointer. Pointers can be declared to point to objects of any data type.

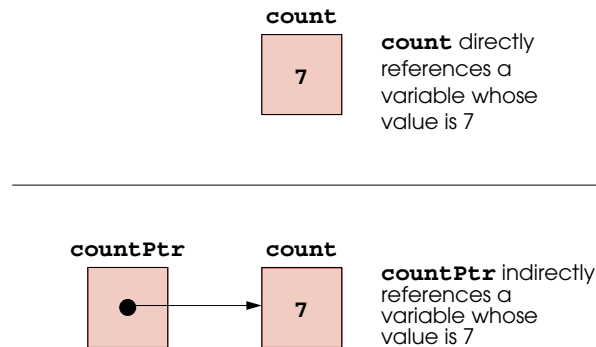


Fig. 7.1 Directly and indirectly referencing a variable.



### Common Programming Error 7.1

The indirection operator (`*`) does not distribute to all variable names in a declaration. Each pointer must be declared with the `*` prefixed to the name.



### Good Programming Practice 7.1

Include the letters `ptr` in pointer variable names to make it clear that these variables are pointers and thus need to 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 `NULL` points to nothing. `NULL` is a symbolic constant defined in the `<stdio.h>` header file (and in several other header files). Initializing a pointer to `0` is equivalent to initializing a pointer to `NULL`, but `NULL` is preferred. When `0` is assigned, it is first converted to a pointer of the appropriate type. The value `0` is the only integer value that can be assigned directly to a pointer variable. Assigning a variable's address to a pointer is discussed in Section 7.3.



### Good Programming Practice 7.2

Initialize pointers to prevent unexpected results.

## 7.3 Pointer Operators

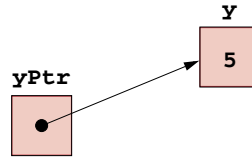
The `&`, or *address operator*, is a unary operator that returns the address of its operand. For example, assuming the declarations

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

the statement

```
yPtr = &y;
```

assigns the address of the variable `y` to pointer variable `yPtr`. Variable `yPtr` is then said to “point to” `y`. Figure 7.2 shows a schematic representation of memory after the preceding assignment is executed.



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

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

The **\*** operator, commonly referred to as the *indirection operator* or *dereferencing operator*, returns the value of the object to which its operand (i.e., a pointer) points. For example, the statement

```
printf( "%d", *yPtr );
```

prints the value of variable **y**, namely 5. Using **\*** in this manner is called *dereferencing a pointer*.



### Common Programming Error 7.2

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

Figure 7.4 demonstrates the pointer operators. The **printf** conversion specification **%p** outputs the memory location as a hexadecimal integer (see Appendix E, “Number Systems,” for more information on hexadecimal integers). Notice that the address of **a** and the value of **aPtr** are identical in the output, thus confirming that the address of **a** is indeed assigned to the pointer variable **aPtr**. The **&** and **\*** operators are complements of one another—when they are both applied consecutively to **aPtr** in either order, the same result is printed. Figure 7.5 lists the precedence and associativity of the operators introduced to this point.



**Fig. 7.3** Representation of **y** and **yPtr** in memory.

```

1  /* Fig. 7.4: fig07_04.c
2     Using the & and * operators */
3  #include <stdio.h>
4
5  int main()
6  {
7     int a;          /* a is an integer */
8     int *aPtr;     /* aPtr is a pointer to an integer */
9
10    a = 7;
11    aPtr = &a;     /* aPtr set to address of a */
12
13    printf( "The address of a is %p"
14           "\nThe value of aPtr is %p", &a, aPtr );
15
16    printf( "\n\nThe value of a is %d"
17           "\nThe value of *aPtr is %d", a, *aPtr );
18
19    printf( "\n\nShowing that * and & are inverses of "
20           "each other.\n&*aPtr = %p"
21           "\n*&aPtr = %p\n", &*aPtr, *&aPtr );
22
23    return 0;
24 }

```

```

The address of a is 0012FF88
The value of aPtr is 0012FF88

The value of a is 7
The value of *aPtr is 7
Proving that * and & are complements of each other.
&*aPtr = 0012FF88
*&aPtr = 0012FF88

```

Fig. 7.4 The & and \* pointer operators.

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

Fig. 7.5 Operator precedence (part 1 of 2).

| Operators        | Associativity | Type        |
|------------------|---------------|-------------|
| ?:               | right to left | conditional |
| = += -= *= /= %= | right to left | assignment  |
| ,                | left to right | comma       |

Fig. 7.5 Operator precedence (part 2 of 2).

## 7.4 Calling Functions by Reference

There are two ways to pass arguments to a function—call by value and call by reference. All function calls in C are call by value. As we saw in Chapter 5, **return** may 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). Many functions require the capability to modify one or more variables in the caller, or to pass a pointer to a large data object to avoid the overhead of passing the object call by value (which, of course, requires making a copy of the object). For these purposes, C provides the capabilities for simulating call by reference.

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

The programs in Fig. 7.6 and Fig. 7.7 present two versions of a function that cubes an integer—**cubeByValue** and **cubeByReference**. Figure 7.6 passes the variable **number** to function **cubeByValue** using call by value (line 12). The **cubeByValue** function cubes its argument and passes the new value back to **main** using a **return** statement. The new value is assigned to **number** in **main**.

Figure 7.7 passes the variable **number** using call by reference (line 14)—the address of **number** is passed—to function **cubeByReference**. Function **cubeByReference** takes a pointer to an **int** called **nPtr** as an argument. The function dereferences the pointer and cubes the value to which **nPtr** points. This changes the value of **number** in **main**. Figures 7.8 and 7.9 analyze graphically the programs in Fig. 7.6 and Fig. 7.7, respectively.



### Common Programming Error 7.3

*Not dereferencing a pointer when it is necessary to do so in order to obtain the value to which the pointer points.*

```

1  /* Fig. 7.6: fig07_06.c
2     Cube a variable using call-by-value */
3  #include <stdio.h>
4

```

Fig. 7.6 Cube a variable using call by value (part 1 of 2).

```

5  int cubeByValue( int ); /* prototype */
6
7  int main()
8  {
9      int number = 5;
10
11     printf( "The original value of number is %d", number );
12     number = cubeByValue( number );
13     printf( "\nThe new value of number is %d\n", number );
14
15     return 0;
16 }
17
18 int cubeByValue( int n )
19 {
20     return n * n * n; /* cube local variable n */
21 }

```

```

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

```

Fig. 7.6 Cube a variable using call by value (part 2 of 2).

```

1  /* Fig. 7.7: fig07_07.c
2     Cube a variable using call-by-reference
3     with a pointer argument */
4
5  #include <stdio.h>
6
7  void cubeByReference( int * ); /* prototype */
8
9  int main()
10 {
11     int number = 5;
12
13     printf( "The original value of number is %d", number );
14     cubeByReference( &number );
15     printf( "\nThe new value of number is %d\n", number );
16
17     return 0;
18 }
19
20 void cubeByReference( int *nPtr )
21 {
22     *nPtr = *nPtr * *nPtr * *nPtr; /* cube number in main */
23 }

```

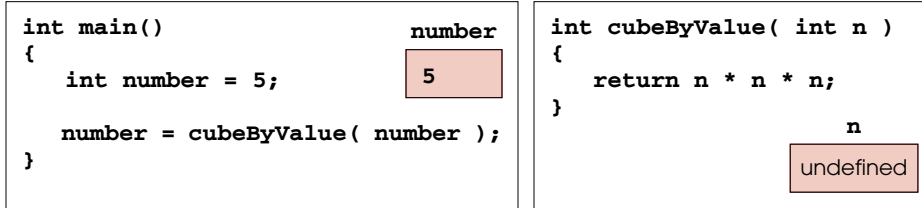
```

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

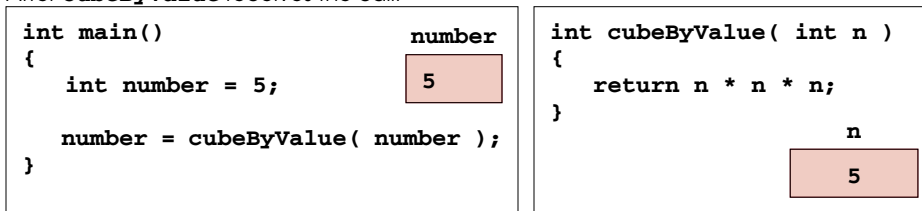
```

Fig. 7.7 Cube a variable using call by reference.

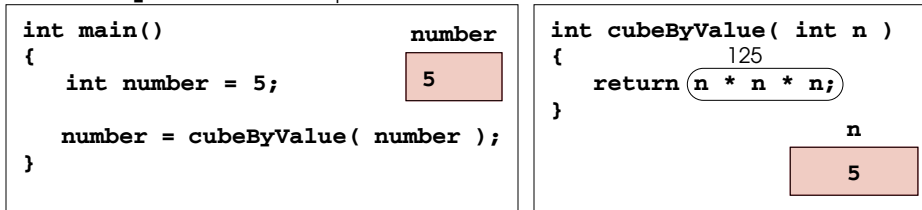
Before `main` calls `cubeByValue`:



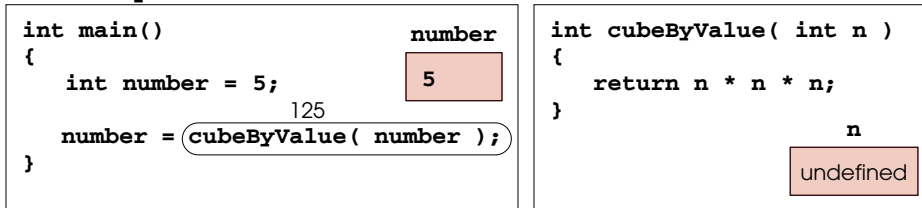
After `cubeByValue` receives the call:



After `cubeByValue` cubes the parameter `n`:



After `cubeByValue` returns to `main`:



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

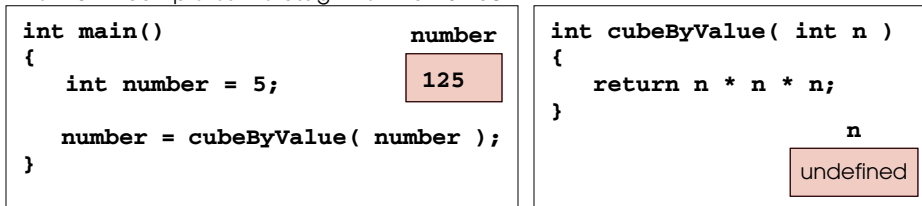
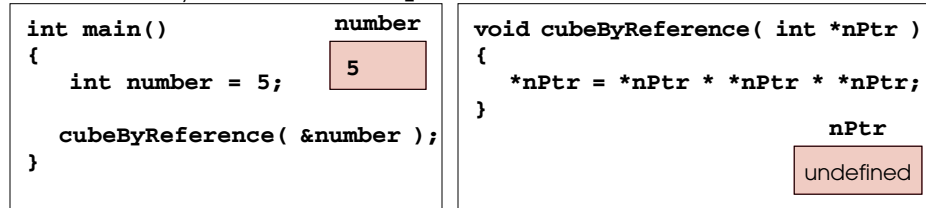
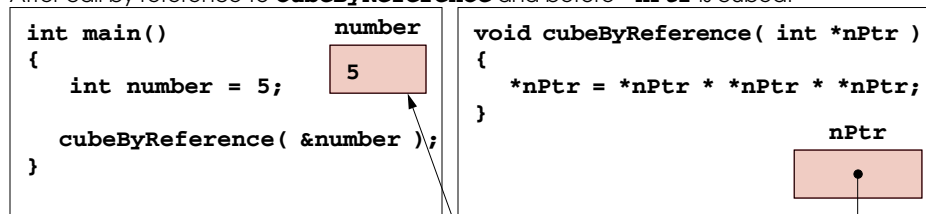


Fig. 7.8 Analysis of a typical call-by-value.

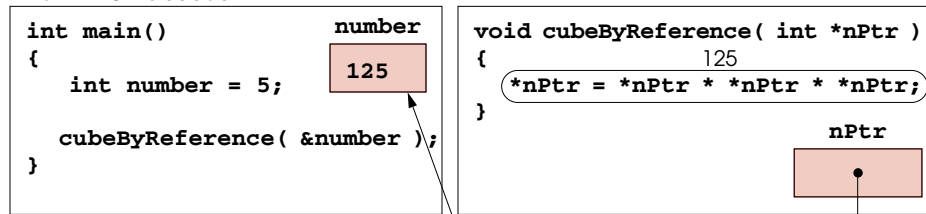
Before the call by reference to `cubeByReference`:



After call by reference to `cubeByReference` and before `*nPtr` is cubed:



After `*nPtr` is cubed:



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

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 20) is:

```
void cubeByReference( int *nPtr )
```

The header specifies that `cubeByReference` receives the address of an integer variable as an argument, stores the address locally in `nPtr`, and does not return a value.

The function prototype for `cubeByReference` contains `int *` in parentheses. As with other variable types, it is not necessary to include names of pointers in function prototypes. Names included for documentation purposes are ignored by the C compiler.

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



### Good Programming Practice 7.3

Use call 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.

## 7.5 Using the `const` Qualifier with Pointers

The `const` qualifier enables the programmer to inform the compiler that the value of a particular variable should not be modified. The `const` qualifier did not exist in early versions of C; it was added to the language by the ANSI C committee.



### Software Engineering Observation 7.1

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



### Portability Tip 7.1

Although `const` is well defined in ANSI C, some systems do not enforce it.

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 tremendous opportunities for improvement in the software engineering of old C code. Also, many programmers currently using ANSI C do not use `const` in their programs because they began programming in early versions of C. These programmers are omitting many opportunities for good software engineering.

Six possibilities exist for using (or not using) `const` with function parameters—two with call-by-value parameter passing and four with call-by-reference parameter passing. How do you choose one of the six possibilities? Let the principle of least privilege be your guide. Always award a function enough access to the data in its parameters to accomplish its specified task, but no more.

In Chapter 5, we explained that all calls in C are call by value—a copy of the argument in the function call is made and passed to the function. If the copy is modified in the function, the original value is maintained without change in the caller. 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 a copy of the original value.

Consider a function that takes a single-subscripted array and its size as arguments and 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 high subscript of the array so the loop can terminate when the printing is completed. The size of the array does not change in the function body.



### Software Engineering Observation 7.2

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

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



### Software Engineering Observation 7.3

Only one value can be altered in a calling function when call by value is used. That value must be assigned from the return value of the function. To modify multiple values in a calling function, call by reference must be used.



### Good Programming Practice 7.4

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



### Common Programming Error 7.4

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

There are four ways to pass a pointer to a function: a non-constant pointer to non-constant data, a constant pointer to non-constant data, a non-constant pointer to constant data, and a constant pointer to constant data. Each of the four combinations provides a different level of access privileges.

The highest level of data access is granted by a non-constant pointer to non-constant data. In this case, the data can be modified through the dereferenced pointer, and the pointer can be modified to point to other data items. A declaration for a non-constant pointer to non-constant data does not include `const`. Such a pointer might be used to receive a string as an argument to a function that uses pointer arithmetic to process (and possibly modify) each character in the string. The function `convertToUpper` of Fig. 7.10 declares its argument, a non-constant pointer to non-constant data called `sPtr` (`char *sPtr`), on line 21. The function processes the array `string` (pointed to by `sPtr`) one character at a time using pointer arithmetic. C standard library function `islower` (called in line 25) tests the character contents of the address pointed to by `sPtr`. If a character is in the range `a` to `z`, `islower` returns true and C standard library function `toupper` (line 26) is called to convert the character to its corresponding uppercase letter; otherwise `islower` returns false and the next character in the string is processed.

---

```

1  /* Fig. 7.10: fig07_10.c
2     Converting lowercase letters to uppercase letters
3     using a non-constant pointer to non-constant data */
4
5  #include <stdio.h>
6  #include <ctype.h>
7
8  void convertToUpper( char * );
9
10 int main()
11 {
12     char string[] = "characters and $32.98";
13

```

---

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

```

14     printf( "The string before conversion is: %s", string );
15     convertToUpper( string );
16     printf( "\nThe string after conversion is: %s\n", string );
17
18     return 0;
19 }
20
21 void convertToUpper( char *sPtr )
22 {
23     while ( *sPtr != '\0' ) {
24
25         if ( islower( *sPtr ) )
26             *sPtr = toupper( *sPtr ); /* convert to uppercase */
27
28         ++sPtr; /* move sPtr to the next character */
29     }
30 }

```

```

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

```

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

A non-constant 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. Such a pointer might be used to receive an array argument to a function that will process each element of the array without modifying the data. For example, the `printCharacters` function of Fig. 7.11 declares parameters `sPtr` to be of type `const char *` (line 23). 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 to output each character in the string until the null character is encountered. After each character is printed, pointer `sPtr` is incremented to point to the next character in the string.

```

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

```

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

---

```

15     printf( "\n" );
16
17     return 0;
18 }
19
20 /* In printCharacters, sPtr is a pointer to a character
21    constant. Characters cannot be modified through sPtr
22    (i.e., sPtr is a "read-only" pointer). */
23 void printCharacters( const char *sPtr )
24 {
25     for ( ; *sPtr != '\0'; sPtr++ ) /* no initialization */
26         printf( "%c", *sPtr );
27 }

```

The string is:  
print characters of a string

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

Figure 7.12 demonstrates the error messages produced by the Borland compiler when attempting to compile a function that receives a non-constant pointer (**xPtr**) to constant data. This function attempts to modify the data pointed to by **xPtr** in line 21—which results in a compilation error.

---

```

1  /* Fig. 7.12: fig07_12.c
2     Attempting to modify data through a
3     non-constant pointer to constant data. */
4
5  #include <stdio.h>
6
7  void f( const int * );
8
9  int main()
10 {
11     int y;
12
13     f( &y ); /* f attempts illegal modification */
14
15     return 0;
16 }
17
18 /* In f, xPtr is a pointer to an integer constant */
19 void f( const int *xPtr )
20 {
21     *xPtr = 100; /* cannot modify a const object */
22 }

```

**Fig. 7.12** Attempting to modify data through a non-constant pointer to constant data (part 1 of 2).

```

FIG07_12.c:
Error E2024 FIG07_12.c 21: Cannot modify a const object in
function f
Warning W8057 FIG07_12.c 22: Parameter 'xPtr' is never used in
function f
*** 1 errors in Compile ***

```

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

As we know, arrays are aggregate data types that store many related data items of the same type under one name. In Chapter 10, we will discuss another form of aggregate data type called a *structure* (sometimes called a *record* in other languages). A structure is capable of storing many related data items of different data types under one name (e.g., storing information about each employee of a company). When a function is called with an array as an argument, the array is automatically passed to the function call by reference. However, structures are always passed call 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 computer’s function call stack. When structure data must be passed to a function, we can use pointers to constant data to get the performance of call by reference and the protection of call by value. When a pointer to a structure is passed, only a copy of the address at which the structure is stored must be made. On a machine with 4-byte addresses, a copy of 4 bytes of memory is made rather than a copy of possibly hundreds or thousands of bytes of the structure.



### Performance Tip 7.1

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

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

A constant pointer to non-constant data is a pointer that always points to the same memory location, and the data at that location can be modified through the pointer. This is the default for an array name. An array name is a constant pointer to the beginning of the array. All data in the array can be accessed and changed by using the array name and array subscripting. A constant pointer to non-constant data can be used to receive an array as an argument to a function that accesses array elements using only array subscript notation. 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). Figure 7.13 attempts to modify a constant pointer. Pointer **ptr** is declared in line 11 to be of type **int \* const**. The declaration 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**. The program attempts to assign the address of **y** to **ptr**, but an error message is generated.

---

```

1  /* Fig. 7.13: fig07_13.c
2     Attempting to modify a constant pointer to
3     non-constant data */
4
5  #include <stdio.h>
6
7  int main()
8  {
9     int x, y;
10
11    int * const ptr = &x; /* ptr is a constant pointer to an
12                          integer. An integer can be modified
13                          through ptr, but ptr always points
14                          to the same memory location. */
15
16    *ptr = 7;
17    ptr = &y;
18
19    return 0;
20 }

```

```

FIG07_13.c:
Error E2024 FIG07_13.c 16: Cannot modify a const object in
function main
*** 1 errors in Compile ***

```

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

The least access privilege is granted by a constant pointer to constant data. Such a pointer always points to the same memory location, and the data at that memory location cannot be modified. This is how an array should be passed to a function that only looks at the array using array subscript notation and does not modify the array. Figure 7.14 declares pointer variable `ptr` to be of type `const int * const` (line 10). This declaration is read from right to left as “`ptr` is a constant pointer to an integer constant.” The figure shows the error messages generated when an attempt is made to modify the data to which `ptr` points, and when an attempt is made to modify the address stored in the pointer variable.

---

```

1  /* Fig. 7.14: fig07_14.c
2     Attempting to modify a constant pointer to
3     constant data. */
4  #include <stdio.h>
5
6  int main()
7  {
8     int x = 5, y;
9
10    const int *const ptr = &x; /* ptr is a constant pointer to a
11                              constant integer. ptr always
12                              points to the same location
13                              and the integer at that
14                              location cannot be modified. */

```

**Fig. 7.14** Attempting to modify a constant pointer to constant data (part 1 of 2).

```

15     printf( "%d\n", *ptr );
16     *ptr = 7;
17     ptr = &y;
18
19     return 0;
20 }

```

```

FIG07_14.c:
Error E2024 FIG07_14.c 16: Cannot modify a const object in
function main
Error E2024 FIG07_14.c 17: Cannot modify a const object in
function main
*** 2 errors in Compile ***

```

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

## 7.6 Bubble Sort Using Call by Reference

Let us modify the bubble sort program of Fig. 6.15 to use two functions—**bubbleSort** and **swap**. Function **bubbleSort** performs the sort of the array. It calls function **swap** to exchange the array elements **array[ j ]** and **array[ j + 1 ]** (see Fig. 7.15). Remember that C enforces information hiding between functions, so **swap** does not have access to individual array elements in **bubbleSort**. Because **bubbleSort** *wants* **swap** to have access to the array elements to be swapped, **bubbleSort** passes each of these elements call by reference to **swap**—the address of each array element is passed explicitly. Although entire arrays are automatically passed call by reference, individual array elements are scalars, and are ordinarily passed call by value. Therefore, **bubbleSort** uses the address operator (**&**) on each of the array elements in the **swap** call (line 39) as follows

```
swap( &array[ j ], &array[ j + 1 ] );
```

to effect call by reference. Function **swap** receives **&array[ j ]** in pointer variable **element1Ptr**. Even though **swap**—because of information hiding—is not allowed to know the name **array[ j ]**, **swap** may use **\*element1Ptr** as a synonym for **array[ j ]**. Therefore, when **swap** references **\*element1Ptr**, it is actually referencing **array[ j ]** in **bubbleSort**. Similarly, when **swap** references **\*element2Ptr**, it is actually referencing **array[ j + 1 ]** in **bubbleSort**. Even though **swap** is not allowed to say

```
temp = array[ j ];
array[ j ] = array[ j + 1 ];
array[ j + 1 ] = temp;
```

precisely the same effect is achieved by lines 44 through 46

```
temp = *element1Ptr;
*element1Ptr = *element2Ptr;
*element2Ptr = temp;
```

in the **swap** function of Fig. 7.15.

```

1  /* Fig. 7.15: fig07_15.c
2     This program puts values into an array, sorts the values into
3     ascending order, and prints the resulting array. */
4  #include <stdio.h>
5  #define SIZE 10
6  void bubbleSort( int *, const int );
7
8  int main()
9  {
10
11     int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
12     int i;
13
14     printf( "Data items in original order\n" );
15
16     for ( i = 0; i < SIZE; i++ )
17         printf( "%4d", a[ i ] );
18
19     bubbleSort( a, SIZE );          /* sort the array */
20     printf( "\nData items in ascending order\n" );
21
22     for ( i = 0; i < SIZE; i++ )
23         printf( "%4d", a[ i ] );
24
25     printf( "\n" );
26
27     return 0;
28 }
29
30 void bubbleSort( int *array, const int size )
31 {
32     void swap( int *, int * );
33     int pass, j;
34     for ( pass = 0; pass < size - 1; pass++ )
35
36         for ( j = 0; j < size - 1; j++ )
37
38             if ( array[ j ] > array[ j + 1 ] )
39                 swap( &array[ j ], &array[ j + 1 ] );
40 }
41
42 void swap( int *element1Ptr, int *element2Ptr )
43 {
44     int hold = *element1Ptr;
45     *element1Ptr = *element2Ptr;
46     *element2Ptr = hold;
47 }

```

```

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

```

Fig. 7.15 Bubble sort with call by reference.

Several features of function `bubbleSort` should be noted. The function header (line 30) 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 size of the array remains fixed during the execution of function `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, it is possible that the sorting algorithm would not run correctly.

The prototype for the function `swap` (line 32) is included in the body of the function `bubbleSort` because it is the only function that calls `swap`. Placing the prototype in `bubbleSort` restricts proper calls of `swap` to those made from `bubbleSort`. Other functions that attempt to call `swap` do not have access to a proper function prototype, so the compiler generates one automatically. This normally results in a prototype that does not match the function header (and generates a compiler error) because the compiler assumes `int` for the return type and the parameter types.



#### Software Engineering Observation 7.4

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

Note that function `bubbleSort` receives the size of the array as a parameter. The function must know the size of the array to sort the array. When an array is passed to a function, the address in memory of the first element of the array is received by the function. The address does not provide any information to the function regarding the number of elements in the array. Therefore, the programmer must provide the function with the array size.

In the program, function `bubbleSort` was explicitly passed the size of the array. There are two main benefits to this approach—software reusability and proper software engineering. By defining the function so it receives the array size as an argument, we enable the function to be used by any program that sorts single-subscripted integer arrays of any size.



#### Software Engineering Observation 7.5

*When passing an array to a function, also pass the size of the array. This helps make the function more general. General functions are often reusable in many programs.*

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



#### Software Engineering Observation 7.6

*Global variables violate the principle of least privilege and are an example of poor software engineering.*



#### Performance Tip 7.2

*Passing the size of an array to a function takes time and requires additional stack space because a copy of the size is made to pass to the function. Global variables, however, require no additional time or space because they can be accessed directly by any function.*

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

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

The number of elements in an array also can be determined at compile time. For example, consider the following array declaration

```
double real[ 22 ];
```

Variables of type **double** normally are stored in 8 bytes of memory. Thus, array **real** contains a total of 176 bytes. To determine the number of elements in the array, the following expression can be used:

```
sizeof( real ) / sizeof( double )
```

---

```

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

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

**Fig. 7.16** The **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 `real`, and divides that value by the number of bytes used in memory to store a `double` value.

Note that function `getSize` returns type `size_t`. Type `size_t` is a type defined by the C standard as the integral type (`unsigned` or `unsigned long`) of the value returned by operator `sizeof`.

Figure 7.17 calculates the number of bytes used to store each of the standard data types.



### Portability Tip 7.2

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

---

```

1  /* Fig. 7.17: fig07_17.c
2     Demonstrating the sizeof operator */
3  #include <stdio.h>
4
5  int main()
6  {
7     char c;
8     short s;
9     int i;
10    long l;
11    float f;
12    double d;
13    long double ld;
14    int array[ 20 ], *ptr = array;
15
16    printf( "      sizeof c = %d"
17           "\tsizeof(char) = %d"
18           "\n      sizeof s = %d"
19           "\tsizeof(short) = %d"
20           "\n      sizeof i = %d"
21           "\tsizeof(int) = %d"
22           "\n      sizeof l = %d"
23           "\tsizeof(long) = %d"
24           "\n      sizeof f = %d"
25           "\tsizeof(float) = %d"
26           "\n      sizeof d = %d"
27           "\tsizeof(double) = %d"
28           "\n      sizeof ld = %d"
29           "\tsizeof(long double) = %d"
30           "\n sizeof array = %d"
31           "\n      sizeof ptr = %d\n",
32           sizeof c, sizeof( char ), sizeof s,
33           sizeof( short ), sizeof i, sizeof( int ),
34           sizeof l, sizeof( long ), sizeof f,
35           sizeof( float ), sizeof d, sizeof( double ),
36           sizeof ld, sizeof( long double ),
37           sizeof array, sizeof ptr );
38
39    return 0;
40 }

```

---

**Fig. 7.17** Using operator `sizeof` 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. 7.17** Using operator **sizeof** to determine standard data type sizes (part 2 of 2).

Operator **sizeof** can be applied to any variable name, type, or constant. When applied to a variable name (that is not an array name) or a constant, the number of bytes used to store the specific type of variable or constant is returned. Note that the parentheses used with **sizeof** are required if a type name is supplied as its operand. Omitting the parentheses results in a syntax error. The parentheses are not required if a variable name is supplied as its operand.

## 7.7 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 in conjunction with pointer variables. This section describes the operators that can have pointers as operands, and how these operators are used.

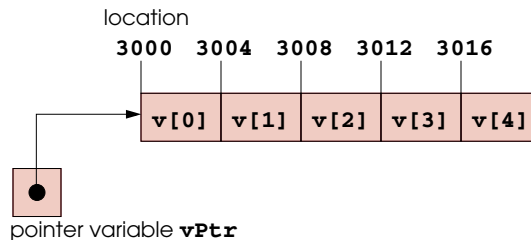
A limited set of arithmetic operations may be performed on pointers. A pointer may be incremented (**++**) or decremented (**--**), an integer may be added to a pointer (**+** or **+=**), an integer may be subtracted from a pointer (**-** or **-=**), or one pointer may be subtracted from another.

Assume that array **int v[ 10 ]** has been declared and its first element is at location **3000** in memory. Assume pointer **vPtr** has been initialized to point to **v[ 0 ]**, i.e., the value of **vPtr** is **3000**. Figure 7.18 diagrams this situation for a machine with 4-byte integers. Note that **vPtr** can be initialized to point to array **v** with either of the statements

```

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

```



**Fig. 7.18** The array **v** and a pointer variable **vPtr** that points to **v**.

**Portability Tip 7.3**

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

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 depend on the object's data type. For example, the statement

```
vPtr += 2;
```

would produce  $3008$  ( $3000 + 2 * 4$ ) assuming an integer is stored in 4 bytes of memory. In the array **v**, **vPtr** would now point to **v[ 2 ]** (Fig. 7.19). If an integer is stored in 2 bytes of memory, then the preceding calculation would result in memory location  $3004$  ( $3000 + 2 * 2$ ). If the array were of a different data type, the preceding statement would increment the pointer by twice the number of bytes that it takes to store an object of that data type. When performing pointer arithmetic on a character array, the results will be consistent with regular arithmetic because each character is one byte long.

If **vPtr** had been incremented to  $3016$ , which points to **v[ 4 ]**, the statement

```
vPtr -= 4;
```

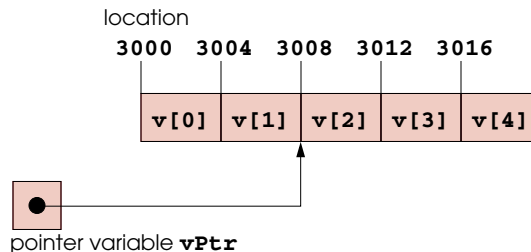
would set **vPtr** back to  $3000$ —the beginning of the array. If a pointer is being incremented or decremented by one, the increment (**++**) and decrement (**--**) operators can be used. Either of the statements

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

increment the pointer to point to the next location in the array. Either of the statements

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

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



**Fig. 7.19** The pointer **vPtr** after pointer arithmetic.

Pointer variables 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 an array. We can not assume that two variables of the same type are stored contiguously in memory unless they are adjacent elements of an array.



#### Common Programming Error 7.5

*Using pointer arithmetic on a pointer that does not refer to an array of values.*



#### Common Programming Error 7.6

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



#### Common Programming Error 7.7

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

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 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 that can represent any pointer type. All pointer types can be assigned a pointer to `void`, and a pointer to `void` can be assigned a pointer of any type. In both cases, a cast operation is not required.

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



#### Common Programming Error 7.8

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



#### Common Programming Error 7.9

*Dereferencing a `void *` pointer.*

Pointers can be compared using equality and relational operators, but such comparisons 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 `NULL`.

## 7.8 The 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.



### Performance Tip 7.3

Array subscripting notation is converted to pointer notation during compilation, so writing array subscripting expressions with pointer notation can save compile time.



### Good Programming Practice 7.5

Use array notation instead of pointer notation when manipulating arrays. Although the program may take slightly longer to compile, it will probably be much clearer.

Assume that integer array `b[ 5 ]` and integer pointer variable `bPtr` have been declared. Since the array name (without a subscript) is a pointer to the first element of the array, we can set `bPtr` equal to the address of the first element in array `b` with the statement

```
bPtr = b;
```

This statement 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 above expression is the *offset* to the pointer. When the pointer points to the beginning of an array, the offset indicates which element of the array should be referenced, and the offset value is identical to the array subscript. The preceding notation is referred to as *pointer/offset notation*. The parentheses are necessary because the precedence of `*` is higher than the precedence of `+`. Without the parentheses, the above expression would add `3` to the value of the expression `*bPtr` (i.e., `3` would be added to `b[ 0 ]` assuming `bPtr` points to the beginning of the array). Just as the array element can be referenced with a pointer expression, the address

```
&b[ 3 ]
```

can be written with the pointer expression

```
bPtr + 3
```

The array itself can be treated as a pointer and used in pointer arithmetic. 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 statement does not modify the array name in any way; `b` still points to the first element in the array.

Pointers can be subscripted exactly as arrays can. For example, the expression

```
bPtr[ 1 ]
```

refers to the array element `b[ 1 ]`. This is referred to as *pointer/subscript notation*.

Remember that an array name is essentially a constant pointer; it always points to the beginning of the array. Thus, the expression

```
b += 3
```

is invalid because it attempts to modify the value of the array name with pointer arithmetic.



### Common Programming Error 7.10

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

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

```

1  /* Fig. 7.20: fig07_20.cpp
2     Using subscripting and pointer notations with arrays */
3
4  #include <stdio.h>
5
6  int main()
7  {
8     int b[] = { 10, 20, 30, 40 };
9     int *bPtr = b; /* set bPtr to point to array b */
10    int i, offset;
11    printf( "Array b printed with:\n"
12           "Array subscript notation\n" );
13
14    for ( i = 0; i < 4; i++ )
15        printf( "b[ %d ] = %d\n", i, b[ i ] );
16
17
18    printf( "\nPointer/offset notation where\n"
19           "the pointer is the array name\n" );
20
21    for ( offset = 0; offset < 4; offset++ )
22        printf( "*( b + %d ) = %d\n", offset, *( b + offset ) );
23
24
25    printf( "\nPointer subscript notation\n" );
26
27    for ( i = 0; i < 4; i++ )
28        printf( " bPtr[ %d ] = %d\n", i, bPtr[ i ] );
29
30    printf( "\nPointer/offset notation\n" );
31
32    for ( offset = 0; offset < 4; offset++ )
33        printf( "*( bPtr + %d ) = %d\n", offset, *( bPtr + offset ) );

```

Fig. 7.20 Using four methods of referencing array elements (part 1 of 2).

```

34
35     return 0;
36 }

```

```

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

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

Pointer subscript notation
bPtr[0] = 10
bPtr[1] = 20
bPtr[2] = 30
bPtr[3] = 40

Pointer/offset notation
*(bPtr + 0) = 10
*(bPtr + 1) = 20
*(bPtr + 2) = 30
*(bPtr + 3) = 40

```

**Fig. 7.20** Using four methods of referencing array elements (part 2 of 2).

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. 7.21. Both functions copy a string (possibly a character array) into a character array. After a comparison of the function prototypes for `copy1` and `copy2`, the functions appear identical. They accomplish the same task; however, they are implemented differently.

```

1  /* Fig. 7.21: fig07_21.cpp
2     Copying a string using array notation
3     and pointer notation. */
4  #include <stdio.h>
5
6  void copy1( char *, const char * );
7  void copy2( char *, const char * );
8
9  int main()
10 {
11     char string1[ 10 ], *string2 = "Hello",
12         string3[ 10 ], string4[] = "Good Bye";

```

**Fig. 7.21** Copying a string using array notation and pointer notation (part 1 of 2).

```

13
14     copy1( string1, string2 );
15     printf( "string1 = %s\n", string1 );
16
17     copy2( string3, string4 );
18     printf( "string3 = %s\n", string3 );
19
20     return 0;
21 }
22
23 /* copy s2 to s1 using array notation */
24 void copy1( char *s1, const char *s2 )
25 {
26     int i;
27     for ( i = 0; ( s1[ i ] = s2[ i ] ) != '\0'; i++ )
28         ; /* do nothing in body */
29 }
30
31 /* copy s2 to s1 using pointer notation */
32 void copy2( char *s1, const char *s2 )
33 {
34     for ( ; ( *s1 = *s2 ) != '\0'; s1++, s2++ )
35         ; /* do nothing in body */
36 }

```

```

string1 = Hello
string3 = Good Bye

```

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

Function **copy1** 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 27) 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** structure, **s1[ i ] = s2[ i ]**, 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 integer value of the null character is zero (false). Remember that the value of an assignment statement is the value assigned to the left argument.

Function **copy2** uses pointers and pointer arithmetic to copy the string in **s2** to the character array **s1**. Again, the **for** structure header (line 34) performs the entire copy operation. The header does not include any variable initialization. As in function **copy1**, the condition (**\*s1 = \*s2**) 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 pointers are incremented to point to the next element of array **s1** and the next character of string **s2**, respectively. When the null character is encountered in **s2**, it is assigned to the dereferenced pointer **s1** and the loop terminates.

Note that the first argument to both **copy1** and **copy2** must be an array large enough to hold the string in the second argument. Otherwise, an error may occur when an attempt

is made to write into a memory location that is not part of the array. Also, note that the second parameter of each function is declared as `const char *` (a constant string). In both functions, the second argument is copied into the first argument—characters are read from it one at a time, but the characters are never modified. Therefore, the second parameter is declared to point to a constant value so the principle of least privilege is enforced. Neither function requires the capability of modifying the second argument, so neither function is provided with that capability.

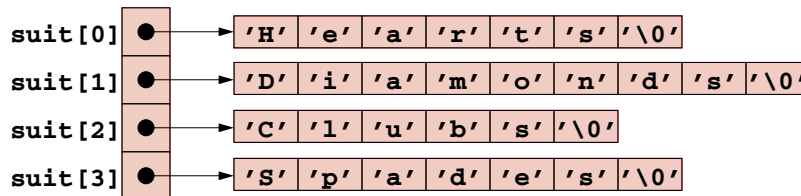
## 7.9 Arrays of Pointers

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 4 elements. The `char *` portion of the declaration indicates that each element of array `suit` is of type “pointer to `char`.” Qualifier `const` indicates that the strings pointed to by each element pointer will not be modified. The four values to be placed in the array are `"Hearts"`, `"Diamonds"`, `"Clubs"` and `"Spades"`. Each 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 7, 9, 6 and 7 characters long, respectively. Although it appears as though these strings are being placed in the `suit` array, only pointers are actually stored in the array (Fig. 7.22). Each pointer points to the first character of its corresponding string. Thus, even though the `suit` array is fixed in size, it provides access to character strings of any length. This flexibility is one example of C’s powerful data structuring capabilities.

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



**Fig. 7.22** A graphical representation of the `suit` array.

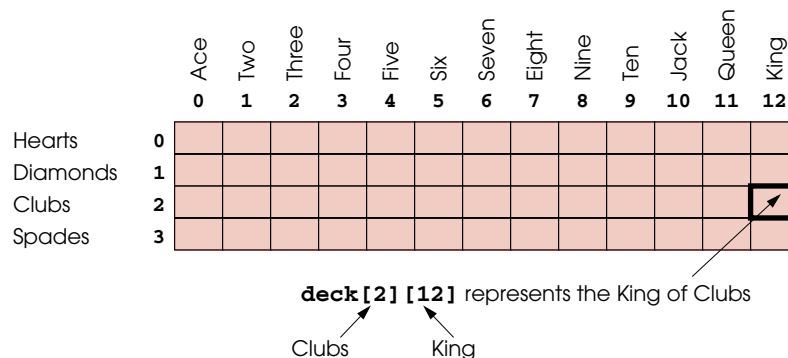
## 7.10 Case Study: A Card Shuffling and Dealing Simulation

In this section, we use random number generation to develop a card shuffling and dealing simulation program. This program can then be used to implement programs that play specific card games. To reveal some subtle performance problems, we have intentionally used sub-optimal shuffling and dealing algorithms. In the exercises and in Chapter 10, we develop more efficient algorithms.

Using the top-down, stepwise refinement approach, we develop a program that will shuffle a deck of 52 playing cards, and then deal each of the 52 cards. 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. 7.23). The rows correspond to the suits—row 0 corresponds to hearts, row 1 to diamonds, row 2 to clubs, and row 3 to spades. The columns correspond to the face values of the cards—columns 0 through 9 correspond to faces ace through ten respectively, and columns 10 through 12 correspond to jack, queen and king. We shall load string array **suit** with character strings representing the four suits, and string array **face** with character strings representing the thirteen face values.

This simulated deck of cards may be shuffled as follows. First the array **deck** is cleared to zeros. Then, a **row** (0–3) and a **column** (0–12) are chosen at random. The number 1 is inserted in array element **deck[ row ][ column ]** to indicate that this card is going to be the first one dealt from the shuffled deck. This process continues with the numbers 2, 3, ..., 52 being randomly inserted in the **deck** array to indicate which cards are to be placed second, third, ..., and fifty-second in the shuffled deck. 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 **rows** and **columns** are repeatedly chosen at random until an unselected card is found. Eventually, the numbers 1 through 52 will occupy the 52 slots of the **deck** array. At this point, the deck of cards is fully shuffled.



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

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



#### Performance Tip 7.4

*Sometimes an algorithm that emerges 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 ] = 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. Note that this program is more efficient if the shuffle and deal portions of the algorithm are combined so each card is dealt as it is placed in the deck. We have chosen to program these operations separately because normally cards are dealt after they are shuffled (not while they are shuffled).

The card shuffling and dealing program is shown in Fig. 7.24 and a sample execution is shown in Fig. 7.25. Note the use of the conversion specifier `%s` to print strings of characters in the calls to `printf`. The corresponding argument in the `printf` call must be a pointer to `char` (or a `char` array). In the `deal` function, the format specification `"%5s of %-8s"` (line 54) prints a character string right-justified in a field of five characters followed by `" of "` and a character string left-justified in a field of eight characters. The minus sign in `%-8s` signifies that the string is left-justified in a field of width 8.

---

```

1  /* Fig. 7.24: fig07_24.c
2     Card shuffling dealing program */
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <time.h>
6

```

---

**Fig. 7.24** Card dealing program (part 1 of 2).

---

```

7 void shuffle( int [][] [ 13 ] );
8 void deal( const int [][] [ 13 ], const char *[], const char *[] );
9
10 int main()
11 {
12     const char *suit[ 4 ] =
13         { "Hearts", "Diamonds", "Clubs", "Spades" };
14     const char *face[ 13 ] =
15         { "Ace", "Deuce", "Three", "Four",
16           "Five", "Six", "Seven", "Eight",
17           "Nine", "Ten", "Jack", "Queen", "King" };
18     int deck[ 4 ][ 13 ] = { 0 };
19
20     srand( time( 0 ) );
21
22     shuffle( deck );
23     deal( deck, face, suit );
24
25     return 0;
26 }
27
28 void shuffle( int wDeck[][] [ 13 ] )
29 {
30     int row, column, card;
31
32     for ( card = 1; card <= 52; card++ ) {
33         do {
34             row = rand() % 4;
35             column = rand() % 13;
36             } while( wDeck[ row ][ column ] != 0 );
37
38             wDeck[ row ][ column ] = card;
39         }
40     }
41
42 void deal( const int wDeck[][] [ 13 ], const char *wFace[],
43           const char *wSuit[] )
44 {
45     int card, row, column;
46
47     for ( card = 1; card <= 52; card++ )
48         for ( row = 0; row <= 3; row++ )
49             for ( column = 0; column <= 12; column++ )
50                 if ( wDeck[ row ][ column ] == card )
51                     printf( "%5s of %-8s%c",
52                             wFace[ column ], wSuit[ row ],
53                             card % 2 == 0 ? '\n' : '\t' );
54 }
55 }
56 }
57 }

```

---

Fig. 7.24 Card dealing program (part 2 of 2).

```

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

```

Fig. 7.25 Sample run of card dealing program.

There is 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 and in a case study in Chapter 10, we correct this deficiency.

## 7.11 Pointers to Functions

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

To illustrate the use of pointers to functions, we have modified the bubble sort program of Fig. 7.15 to form the program of Fig. 7.26. Our new program consists of **main**, and the functions **bubble**, **swap**, **ascending** and **descending**. Function **bubbleSort** receives a pointer to a function—either function **ascending** or function **descending**—as an argument in addition to an integer array and the size of the array. The program prompts the user to choose if 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** causing the array to be sorted into increasing order. If the user enters **2**, a pointer to function

**descending** is passed to function **bubble** causing the array to be sorted into decreasing order. The output of the program is shown in Fig. 7.27.

The following parameter appears in the function header for **bubble** (line 42)

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

This tells **bubble** to expect a parameter that is a pointer to a function that receives two integer parameters and returns an integer result. Parentheses are needed around **\*compare** because **\*** has a lower precedence than the parentheses enclosing the function parameters. If we had not included the parentheses, the declaration would have been

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

---

```

1  /* Fig. 7.26: fig07_26.cpp
2     Multipurpose sorting program using function pointers */
3  #include <stdio.h>
4  #define SIZE 10
5  void bubble( int [], const int, int (*)( int, int ) );
6  int ascending( int, int );
7  int descending( int, int );
8
9  int main()
10 {
11
12     int order,
13         counter,
14         a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
15
16     printf( "Enter 1 to sort in ascending order,\n"
17            "Enter 2 to sort in descending order: " );
18     scanf( "%d", &order );
19     printf( "\nData items in original order\n" );
20
21     for ( counter = 0; counter < SIZE; counter++ )
22         printf( "%5d", a[ counter ] );
23
24     if ( order == 1 ) {
25         bubble( a, SIZE, ascending );
26         printf( "\nData items in ascending order\n" );
27     }
28     else {
29         bubble( a, SIZE, descending );
30         printf( "\nData items in descending order\n" );
31     }
32
33     for ( counter = 0; counter < SIZE; counter++ )
34         printf( "%5d", a[ counter ] );
35
36     printf( "\n" );
37
38     return 0;
39 }
```

---

Fig. 7.26 Multipurpose sorting program using function pointers (part 1 of 2).

```

40
41 void bubble( int work[], const int size,
42             int (*compare)( int, int ) )
43 {
44     int pass, count;
45
46     void swap( int *, int * );
47
48     for ( pass = 1; pass < size; pass++ )
49
50         for ( count = 0; count < size - 1; count++ )
51
52             if ( (*compare)( work[ count ], work[ count + 1 ] ) )
53                 swap( &work[ count ], &work[ count + 1 ] );
54 }
55
56 void swap( int *element1Ptr, int *element2Ptr )
57 {
58     int temp;
59
60     temp = *element1Ptr;
61     *element1Ptr = *element2Ptr;
62     *element2Ptr = temp;
63 }
64
65 int ascending( int a, int b )
66 {
67     return b < a;    /* swap if b is less than a */
68 }
69
70 int descending( int a, int b )
71 {
72     return b > a;    /* swap if b is greater than a */
73 }

```

Fig. 7.26 Multipurpose sorting program using function pointers (part 2 of 2).

```

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

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

```

```

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

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

```

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

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

The corresponding parameter in the function prototype of **bubble** (line 5) is

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

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

The function passed to **bubble** is called in an **if** statement (line 52) as follows

```
if ( (*compare)( work[ count ], work[ count + 1 ] ) )
```

Just as a pointer to a variable is dereferenced to access the value of the variable, a pointer to a function is dereferenced to use the function.

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

```
if ( compare( work[ count ], work[ count + 1 ] ) )
```

which uses the pointer directly as the function name. We prefer the first method of calling a function through a pointer because it explicitly illustrates that **compare** is a pointer to a function that is dereferenced to call the function. The second method of calling a function through a pointer makes it appear as though **compare** is an actual function. This may be confusing to a user of the program who would like to see the definition of function **compare** and finds that it is never defined in the file.

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

Figure 7.28 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 return nothing. Pointers to these three functions are stored in array **f** which is declared (line 10) as follows:

```
void ( *f[3] )( int ) = { function1, function2, function3 };
```

The declaration is read beginning in the leftmost set of parentheses, “**f** is an array of 3 pointers to functions that take an **int** as an argument and that return **void**.” The array is initialized with the names of the three functions. When the user enters a value between 0 and 2, the value is used as the subscript into the array of pointers to functions. The function call (line 17) is made as follows:

```
(*f[ choice ])( choice );
```

In the function 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 demonstrate that the function is called correctly. In the exercises, you will develop a menu-driven system.

```
1  /* Fig. 7.28: fig07_28.cpp
2     Demonstrating an array of pointers to functions */
3  #include <stdio.h>
4  void function1( int );
5  void function2( int );
6  void function3( int );
7
8  int main()
9  {
10     void (*f[ 3 ])( int ) = { function1, function2, function3 };
11     int choice;
12
13     printf( "Enter a number between 0 and 2, 3 to end: " );
14     scanf( "%d", &choice );
15
16     while ( choice >= 0 && choice < 3 ) {
17         (*f[ choice ])( choice );
18         printf( "Enter a number between 0 and 2, 3 to end: " );
19         scanf( "%d", &choice );
20     }
21
22     printf( "Program execution completed.\n" );
23
24     return 0;
25 }
26
27 void function1( int a )
28 {
29     printf( "You entered %d"
30           " so function1 was called\n\n", a );
31 }
32
33 void function2( int b )
34 {
35     printf( "You entered %d"
36           " so function2 was called\n\n", b );
37 }
38
39 void function3( int c )
40 {
41     printf( "You entered %d"
42           " so function3 was called\n\n", c );
43 }
```

```
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
```

**Fig. 7.28** Demonstrating an array of pointers to functions (part 1 of 2).

```

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
You entered 3 to end

```

**Fig. 7.28** Demonstrating an array of pointers to functions (part 2 of 2).

### SUMMARY

- Pointers are variables that contain as their values addresses of other variables.
- Pointers must be declared before they can be used.
- The declaration

```
int *ptr;
```

declares **ptr** to be a pointer to an object 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. Initializing a pointer to **0** and initializing that same pointer to **NULL** are identical.
- The only integer that can be assigned to a pointer is **0**.
- The **&** (address) operator returns the address of its operand.
- The operand of the address operator must be a variable; the address operator cannot be applied to constants, to expressions, or to variables declared with the storage class **register**.
- The **\*** operator, referred to as the indirection or dereferencing operator, returns the value 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 is passed. The called function then uses the indirection operator (**\***) to modify the value of the argument in the calling function.
- A function receiving an address as an argument must include a pointer as its corresponding formal parameter.
- It is not necessary to include the names of pointers in function prototypes; it is only necessary to include the pointer types. Pointer names may be included for documentation reasons, but the compiler ignores them.
- The **const** qualifier enables the programmer to inform the compiler that the value of a particular variable should not be modified.
- If an attempt is made to modify a value that is declared **const**, the compiler catches it and issues either a warning or an error depending on the particular compiler.
- There are four ways to pass a pointer to a function: a non-constant pointer to non-constant data, a constant pointer to non-constant data, a non-constant pointer to constant data, and a constant pointer to constant data.
- Arrays are automatically passed by reference because the value of the array name is the address of the array.
- To pass a single element of an array call by reference, the address of the specific array element must be passed.
- C provides the special unary operator **sizeof** to determine the size in bytes of an array (or any other data type) during program compilation.

- When applied to the name of an array, the **sizeof** operator returns the total number of bytes in the array as an integer.
- Operator **sizeof** can be applied to any variable name, type, or constant.
- Type **size\_t** is a type defined by the standard as the integral type (**unsigned** or **unsigned long**) of the value returned by operator **sizeof**.
- The arithmetic operations that may be performed on pointers are incrementing (**++**) a pointer, decrementing (**--**) a pointer, adding (**+** or **+=**) a pointer and an integer, subtracting (**-** or **-=**) a pointer and an integer, 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 pointed to.
- 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.
- When performing pointer arithmetic on a character array, the results are like regular arithmetic because each character is stored in one byte of 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 pointer to **void** which is a generic pointer type that can hold pointers of any type. Pointers to **void** can be assigned pointers of other types and can be assigned to pointers of other types without a cast.
- A pointer to **void** may not be dereferenced.
- Pointers can be compared using the equality and relational operators. Pointer comparisons are normally meaningful only if the pointers point to members of the same array.
- Pointers can be subscripted exactly as array names can.
- An array name without a subscript is a pointer to the first element of the array.
- In pointer/offset notation, 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 a separate pointer that points to the array.
- An array name is a constant pointer that always points to the same location in memory. Array names cannot be modified as conventional pointers can.
- It is possible to have arrays of pointers.
- It is possible to have pointers to functions.
- 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.

## TERMINOLOGY

adding a pointer and an integer

address operator (**&**)

array of pointers

array of strings

call by reference

call by value

character pointer

**const**

constant pointer

constant pointer to constant data

constant pointer to non-constant data

decrement a pointer

dereference a pointer

dereferencing operator (**\***)

directly reference a variable

dynamic memory allocation

function pointer

increment a pointer

|   |  |
|---|--|
| indefinite postponement                   | pointer indexing                         |
| indirection                               | pointer/offset notation                  |
| indirection operator (*)                  | pointer subscripting                     |
| indirectly reference a variable           | pointer to a function                    |
| initializing pointers                     | pointer to <b>void</b> ( <b>void *</b> ) |
| linked list                               | pointer types                            |
| non-constant pointer to constant data     | principle of least privilege             |
| non-constant pointer to non-constant data | simulated call by reference              |
| <b>NULL</b> pointer                       | <b>sizeof</b>                            |
| offset                                    | <b>size_t</b> type                       |
| pointer                                   | string array                             |
| pointer arithmetic                        | subtracting an integer from a pointer    |
| pointer assignment                        | subtracting two pointers                 |
| pointer comparison                        | top-down, stepwise refinement            |
| pointer expression                        | <b>void *</b> (pointer to <b>void</b> )  |

### COMMON PROGRAMMING ERRORS

- 7.1** The indirection operator (\*) does not distribute to all variable names in a declaration. Each pointer must be declared with the \* prefixed to the name.
- 7.2** Dereferencing a pointer that has not been properly initialized or that has not been assigned to point to a specific location in memory. This could cause a fatal execution time error, or it could accidentally modify important data and allow the program to run to completion providing incorrect results.
- 7.3** Not dereferencing a pointer when it is necessary to do so in order to obtain the value to which the pointer points.
- 7.4** Being unaware that a function is expecting pointers as arguments for call by reference and passing arguments call by value. Some compilers take the values assuming they are pointers and dereference the values as pointers. At run-time, memory access violations or segmentation faults are often generated. Other compilers catch the mismatch in types between arguments and parameters and generate error messages.
- 7.5** Using pointer arithmetic on a pointer that does not refer to an array of values.
- 7.6** Subtracting or comparing two pointers that do not refer to the same array.
- 7.7** Running off either end of an array when using pointer arithmetic.
- 7.8** Assigning a pointer of one type to a pointer of another type if neither is of type **void \*** causes a syntax error.
- 7.9** Dereferencing a **void \*** pointer.
- 7.10** Attempting to modify an array name with pointer arithmetic is a syntax error.

### GOOD PROGRAMMING PRACTICES

- 7.1** Include the letters **ptr** in pointer variable names to make it clear that these variables are pointers and thus need to be handled appropriately.
- 7.2** Initialize pointers to prevent unexpected results.
- 7.3** Use call 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.
- 7.4** Before using a function, check the function prototype for the function to determine if the function is able to modify the values passed to it.

- 7.5 Use array notation instead of pointer notation when manipulating arrays. Although the program may take slightly longer to compile, it will probably be much clearer.

### PERFORMANCE TIPS

- 7.1 Pass large objects such as structures using pointers to constant data to obtain the performance benefits of call by reference and the security of call by value.
- 7.2 Passing the size of an array to a function takes time and requires additional stack space because a copy of the size is made to pass to the function. Global variables, however, require no additional time or space because they can be accessed directly by any function.
- 7.3 Array subscripting notation is converted to pointer notation during compilation, so writing array subscripting expressions with pointer notation can save compile time.
- 7.4 Sometimes an algorithm that emerges in a “natural” way can contain subtle performance problems such as indefinite postponement. Seek algorithms that avoid indefinite postponement.

### PORTABILITY TIPS

- 7.1 Although **const** is well defined in ANSI C, some systems do not enforce it.
- 7.2 The number of bytes used to store a particular data type may vary between systems. When writing programs that depend on data type sizes, and that will run on several computer systems, be sure to use **sizeof** to determine the number of bytes used to store the data types.
- 7.3 Most computers today have 2-byte or 4-byte integers. Some of the newer machines use 8-byte integers. Because the results of pointer arithmetic depend on the size of the objects a pointer points to, pointer arithmetic is machine dependent.

### SOFTWARE ENGINEERING OBSERVATIONS

- 7.1 The **const** qualifier can be used to enforce the principle of least privilege. Using the principle of least privilege to properly design software tremendously reduces debugging time and improper side effects and makes a program easier to modify and maintain.
- 7.2 If a value does not (or should not) change in the body of a function to which it is passed, the value should be declared **const** to ensure that it is not accidentally modified.
- 7.3 Only one value can be altered in a calling function when call by value is used. That value must be assigned from the return value of the function. To modify multiple values in a calling function, call by reference must be used.
- 7.4 Placing function prototypes in the definitions of other functions enforces the principle of least privilege by restricting proper function calls to the functions in which the prototypes appear.
- 7.5 When passing an array to a function, also pass the size of the array. This helps make the function more general. General functions are often reusable in many programs.
- 7.6 Global variables violate the principle of least privilege and are an example of poor software engineering.

### SELF-REVIEW EXERCISES

- 7.1 Answer each of the following:
- A pointer is a variable that contains as its value the \_\_\_\_\_ of another variable.
  - The three values that can be used to initialize a pointer are \_\_\_\_\_, \_\_\_\_\_ or an \_\_\_\_\_.
  - The only integer that can be assigned to a pointer is \_\_\_\_\_.
- 7.2 State whether the following are *true* or *false*. If the answer is *false*, explain why.

- a) The address operator (**&**) can only be applied to constants, to expressions, and to variables declared with the storage class **register**.
- b) A pointer that is declared to be **void** can be dereferenced.
- c) Pointers of different types may not be assigned to one another without a cast operation.

**7.3** Answer each of the following. Assume that single-precision floating-point numbers are stored in 4 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 **float** called **numbers** with 10 elements, and initialize the elements to the values **0.0, 1.1, 2.2, ..., 9.9**. Assume the symbolic constant **SIZE** has been defined as **10**.
- b) Declare a pointer **nPtr** that points to an object of type **float**.
- c) Print the elements of array **numbers** using array subscript notation. Use a **for** structure, and assume the integer control variable **i** has been declared. Print each number with 1 position of precision to the right of the decimal point.
- d) Give two separate statements that assign the starting address of array **numbers** to the pointer variable **nPtr**.
- e) Print the elements of array **numbers** using pointer/offset notation with the pointer **nPtr**.
- f) Print the elements of array **numbers** using pointer/offset notation with the array name as the pointer.
- g) Print the elements of array **numbers** by subscripting pointer **nPtr**.
- h) Refer to element 4 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 -- 4**. What is the value stored at that location?

**7.4** For each of the following, write a single statement that performs the indicated task. Assume that floating-point variables **number1** and **number2** have been declared and that **number1** has been initialized to **7.3**.

- a) Declare the variable **fPtr** to be a pointer to an object of type **float**.
- 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**. Use the **%p** conversion specifier.
- g) Print the address stored in **fPtr**. Use the **%p** conversion specifier. Is the value printed the same as the address of **number1**?

**7.5** Do each of the following:

- a) Write the function header for a function called **exchange** that takes two pointers to 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).

**7.6** Find the error in each of the following program segments. Assume

```
int *zPtr;    /* zPtr will reference array z */
int *aPtr = NULL;
```

```

void *sPtr = NULL;
int number, i;
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 ( i = 0; i <= 5; i++ )
       printf( "%d ", zPtr[ i ] );
e) /* assign the value pointed to by sPtr to number */
   number = *sPtr;
f) ++z;

```

### ANSWERS TO SELF-REVIEW EXERCISES

- 7.1** a) address. b) **0, NULL**, an address. c) **0**.
- 7.2** a) False. The address operator can only be applied to variables, and it cannot be applied to variables declared with storage class **register**.  
 b) False. A pointer to **void** cannot be dereferenced because there is no way to know exactly how many bytes of memory should be dereferenced.  
 c) False. Pointers of type **void** can be assigned pointers of other types, and pointers of type **void** can be assigned to pointers of other types.
- 7.3** a) `float numbers[ SIZE ] = { 0.0, 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9 };`  
 b) `float *nPtr;`  
 c) `for ( i = 0; i <= SIZE - 1; i++ )  
     printf( "%.1f ", numbers[ i ] );`  
 d) `nPtr = numbers;  
     nPtr = &numbers[ 0 ];`  
 e) `for ( i = 0; i <= SIZE - 1; i++ )  
     printf( "%.1f ", *( nPtr + i ) );`  
 f) `for ( i = 0; i <= SIZE - 1; i++ )  
     printf( "%.1f ", *( numbers + i ) );`  
 g) `for ( i = 0; i <= SIZE - 1; i++ )  
     printf( "%.1f ", nPtr[ i ] );`  
 h) `numbers[ 4 ]  
     *( numbers + 4 )  
     nPtr[ 4 ]  
     *( nPtr + 4 )`  
 i) The address is  $1002500 + 8 * 4 = 1002532$ . The value is **8.8**.  
 j) The address of `numbers[ 5 ]` is  $1002500 + 5 * 4 = 1002520$ .  
 The address of `nPtr - 4` is  $1002520 - 4 * 4 = 1002504$ .  
 The value at that location is **1.1**.
- 7.4** a) `float *fPtr;`  
 b) `fPtr = &number1;`  
 c) `printf( "The value of *fPtr is %f\n", *fPtr );`  
 d) `number2 = *fPtr;`

- e) `printf( "The value of number2 is %f\n", number2 );`
- f) `printf( "The address of number1 is %p\n", &number1 );`
- g) `printf( "The address stored in fptr is %p\n", fPtr );`  
Yes, the value is the same.

- 7.5**
- a) `void exchange( float *x, float *y )`
  - b) `void exchange( float *, float * );`
  - c) `int evaluate( int x, int (*poly)( int ) )`
  - d) `int evaluate( int, int (*)( int ) );`

- 7.6**
- 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 final value of the control variable in the `for` structure to `4`.
  - e) Error: Dereferencing a void pointer.  
Correction: In order to dereference the pointer, it must first be cast to an integer pointer.  
Change the above 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.

## EXERCISES

- 7.7** Answer each of the following:
- a) The \_\_\_\_\_ operator returns the location in memory where its operand is stored.
  - b) The \_\_\_\_\_ operator returns the value of the object to which its operand points.
  - c) To simulate call by reference when passing a non-array variable to a function, it is necessary to pass the \_\_\_\_\_ of the variable to the function.
- 7.8** 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 may be manipulated in precisely the same manner as pointers.
- 7.9** Answer each of the following. Assume that unsigned integers are stored in 2 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 5 elements, and initialize the elements to the even integers from 2 to 10. Assume the symbolic constant `SIZE` has been defined as `5`.
  - b) Declare a pointer `vPtr` that points to an object of type `unsigned int`.
  - c) Print the elements of array `values` using array subscript notation. Use a `for` structure, and assume integer control variable `i` has been declared.
  - d) Give two separate statements that assign the starting address of array `values` to pointer variable `vPtr`.
  - e) Print the elements of array `values` using pointer/offset notation.
  - f) Print the elements of array `values` using pointer/offset notation with the array name as the pointer.
  - g) Print the elements of array `values` by subscripting the pointer to the array.
  - h) Refer to element 5 of array `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 `vPtr` points to `values[4]`, what address is referenced by `vPtr -= 4`. What value is stored at that location?

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

- a) Declare the variable `lPtr` to be a pointer to an object of type `long`.
- b) Assign the address of variable `value1` to pointer variable `lPtr`.
- c) Print the value of the object pointed to by `lPtr`.
- d) Assign the value of the object pointed to by `lPtr` to variable `value2`.
- e) Print the value of `value2`.
- f) Print the address of `value1`.
- g) Print the address stored in `lPtr`. Is the value printed the same as the address of `value1`?

**7.11** Do each of the following.

- a) Write the function header for function `zero` which 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` which takes an integer array parameter `oneTooSmall` and returns an integer.
- d) Write the function prototype for the function described in part (c).

*Note: Exercises 7.12 through 7.15 are reasonably challenging. Once you have done these problems, you ought to be able to implement most popular card games easily.*

**7.12** Modify the program in Fig. 7.24 so that the card dealing function deals a five card poker hand. Then write the following additional functions:

- a) Determine if the hand contains a pair.
- b) Determine if the hand contains two pairs.
- c) Determine if the hand contains three of a kind (e.g., three jacks).
- d) Determine if the hand contains four of a kind (e.g., four aces).
- e) Determine if the hand contains a flush (i.e., all five cards of the same suit).
- f) Determine if the hand contains a straight (i.e., five cards of consecutive face values).

**7.13** Use the functions developed in Exercise 7.12 to write a program that deals two five-card poker hands, evaluates each hand, and determines which is the better hand.

**7.14** Modify the program developed in Exercise 7.13 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!)

**7.15** Modify the program developed in Exercise 7.14 so that it can handle the dealer's hand automatically, 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 (this, too, is a difficult problem). Play 20 more games. Does your modified program play a better game?

**7.16** In the card shuffling and dealing program of Fig. 7.24, 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 the program of Fig. 7.24 as follows. Begin by initializing the **deck** array as shown in Fig. 7.29. 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 if the deck is satisfactorily shuffled (as in Fig. 7.30, 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. 7.24 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. In Chapter 10 we develop a dealing algorithm that requires only one operation per card.

**7.17** (*Simulation: The Tortoise and the Hare*) In this problem you will recreate one of the truly great moments in history, namely 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.

| Unshuffled <b>deck</b> 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. 7.29** Unshuffled **deck** array.

| Sample shuffled <b>deck</b> 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. 7.30** Sample shuffled **deck** array.

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 following rules:

| 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  |

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  $1 \leq i \leq 10$ . For the tortoise, perform a “fast plod” when  $1 \leq i \leq 5$ , a “slip” when  $6 \leq i \leq 7$ , or a “slow plod” when  $8 \leq i \leq 10$ . Use a similar technique to move the hare.

Begin the race by printing

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

Then, for each tick of the clock (i.e., each repetition of a loop), print a 70 position line showing the letter **T** in the position of the tortoise and the letter **H** in the position of the hare. Occasionally, the contenders will 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 each line is printed, test if either animal has reached or passed square 70. If so, then 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 tick of the clock, 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 at how involved your 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 (through the technique of software-based *simulation*) on which you can execute your machine language programs!

**7.18** (*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**.

Each instruction written in SML occupies one word of the Simpletron's memory (and hence instructions are signed four-digit decimal numbers). We shall 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 either 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*, which specifies the operation to be performed. SML operation codes are summarized in Fig. 7.31.

| Operation code                         | Meaning  |
|--|--|
| <i>Input/output operations:</i>        |  |
| <b>#define READ 10</b>                 | Read a word from the terminal into a specific location in memory.  |
| <b>#define WRITE 11</b>                | Write a word from a specific location in memory to the terminal.   |
| <i>Load/store operations:</i>          |  |
| <b>#define LOAD 20</b>                 | Load a word from a specific location in memory into the accumulator.   |
| <b>#define STORE 21</b>                | Store a word from the accumulator into a specific location in memory.  |
| <i>Arithmetic operations:</i>          |  |
| <b>#define ADD 30</b>                  | Add a word from a specific location in memory to the word in the accumulator (leave result in accumulator).        |
| <b>#define SUBTRACT 31</b>             | Subtract a word from a specific location in memory from the word in the accumulator (leave result in accumulator). |
| <b>#define DIVIDE 32</b>               | Divide a word from a specific location in memory into the word in the accumulator (leave result in accumulator).   |
| <b>#define MULTIPLY 33</b>             | Multiply a word from a specific location in memory by the word in the accumulator (leave result in accumulator).   |
| <i>Transfer of control operations:</i> |  |
| <b>#define BRANCH 40</b>               | Branch to a specific location in memory.   |
| <b>#define BRANCHNEG 41</b>            | Branch to a specific location in memory if the accumulator is negative.  |
| <b>#define BRANCHZERO 42</b>           | Branch to a specific location in memory if the accumulator is zero.  |
| <b>#define HALT 43</b>                 | Halt, i.e., the program has completed its task.  |

**Fig. 7.31** Simpletron Machine Language (SML) operation codes.

The last two digits of an SML instruction are the *operand*, which is the address of the memory location containing the word to which the operation applies. Now let us consider several simple SML programs.

| Example 1<br>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)   |

This SML program 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). Then **+1008** reads the next number into location **08**. The *load* instruction, **+2007**, puts 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 the result back into memory location **09** from which the *write* instruction, **+1109**, takes the number and prints it (as a signed four-digit decimal number). The *halt* instruction, **+4300**, terminates execution.

| Example 2<br>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)            |

This SML program reads two numbers from the keyboard and 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 10 positive numbers and compute and print their sum.
- 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.

**7.19** (*A 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 will actually be able to run, test, and debug the SML programs you wrote in Exercise 7.18.

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. ***
```

Simulate the memory of the Simpletron with a single-subscripted array **memory** that has 100 elements. Now assume that the simulator is running, and let us examine the dialog as we enter the program of Example 2 of Exercise 7.18:

```
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 ***
```

The SML program has now been placed (or loaded) into the 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 the variable **accumulator** to represent the accumulator register. Use the variable **instructionCounter** to keep track of the location in memory that contains the instruction being performed. Use the variable **operationCode** to indicate the operation currently being performed, i.e., the left two digits of the instruction word. Use the 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 the variable **operationCode**, and “pick off” the right two digits and place them in **operand**.

When Simpletron begins execution, the special registers are initialized as follows:

```

accumulator          +0000
instructionCounter    00
instructionRegister   +0000
operationCode        00
operand              00

```

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 **instructionCounter** 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[ instructionCounter ];
```

The operation code and the 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 twelve 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:    scanf( "%d", &memory[ operand ] );
load:    accumulator = memory[ operand ];
add:     accumulator += memory[ operand ];
Various branch instructions: We'll discuss these shortly.
halt:    This instruction prints the message

```

```
*** Simpletron execution terminated ***
```

and then prints 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. 7.32. Note that a dump after executing a Simpletron program would show the actual values of instructions and data values at the moment execution terminated.

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

```
scanf( "%d", &memory[ operand ] );
```

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

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

```
++instructionCounter;
```

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 be executed.

| REGISTERS:          |       |       |       |       |       |       |       |       |       |       |
|---------------------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| accumulator         |       |       |       |       |       |       |       |       |       | +0000 |
| instructionCounter  |       |       |       |       |       |       |       |       |       | 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. 7.32 A sample dump.

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

```
instructionCounter = operand;
```

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

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

At this point you should implement your Simpletron simulator and run each of the SML programs you wrote in Exercise 7.18. 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.

**7.20** Modify the card shuffling and dealing program of Fig. 7.24 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. 7.24.

**7.21** What does this program do?

---

```

1  /* ex07_21.c */
2  #include <stdio.h>
3
4  void mystery1( char *, const char * );
5
6  int main()
7  {
8      char string1[ 80 ], string2[ 80 ];
9
10     printf( "Enter two strings: " );
11     scanf( "%s%s" , string1, string2 );
12     mystery1( string1, string2 );
13     printf( " %s", string1 );
14
15     return 0;
16 }
17
18 void mystery1( char *s1, const char *s2 )
19 {
20     while ( *s1 != '\0' )
21         ++s1;
22
23     for ( ; *s1 = *s2; s1++, s2++ )
24         ; /* empty statement */
25 }

```

---

**7.22** What does this program do?

---

```

1  /* ex07_22.c */
2  #include <stdio.h>
3
4  int mystery2( const char * );
5
6  int main()
7  {
8      char string[ 80 ];
9
10     printf( " Enter a string: ");
11     scanf( "%s", string );
12     printf( " %d\n", mystery2( string ) );
13
14     return 0;
15 }
16
17 int mystery2( const char *s )
18 {
19     int x;

```

---

---

```

20
21     for ( x = 0; *s != '\0'; s++ )
22         ++x;
23
24     return x;
25 }

```

---

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

- `int *number;`  
`printf( "%d\n", *number );`
- `float *realPtr;`  
`long *integerPtr;`  
`integerPtr = realPtr;`
- `int * x, y;`  
`x = y;`
- `char s[] = "this is a character array";`  
`int count;`  
`for ( ; *s != '\0'; s++)`  
 `printf( "%c ", *s );`
- `short *numPtr, result;`  
`void *genericPtr = numPtr;`  
`result = *genericPtr + 7;`
- `float x = 19.34;`  
`float xPtr = &x;`  
`printf( "%f\n", xPtr );`
- `char *s;`  
`printf( "%s\n", s );`

**7.24** (*Quicksort*) In the examples and exercises of Chapter 6, we discussed the sorting techniques of 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:

- 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.
- 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, it 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

- 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 new array is

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

Element 12 is in *italic* 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 new array is

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 new array is

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 on 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.

**7.25** (*Maze Traversal*) The following grid of ones and zeros is a double-subscripted array representation of a maze.

```

# # # # # # # # # # #
# . . . # . . . . . #
. . # . # . # # # . #
# # # . # . . . # . #
# . . . . # # # . # . .
# # # # . # . # . # . #
# . . # . # . # . # . #
# # . # . # . # . # . #
# . . . . . . . # . #
# # # # # # . # # # . #
# . . . . . # . . . #
# # # # # # # # # # #

```

The ones represent the walls of the maze, and the zeros represent squares in the possible paths through the maze.

There is a simple algorithm for walking through a maze that guarantees finding the exit (assuming 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.

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.

**7.26** (*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 7.25 using several randomly generated mazes.

**7.27** (*Mazes of Any Size*) Generalize functions **mazeTraverse** and **mazeGenerator** of Exercises 7.25 and 7.26 to process mazes of any width and height.

**7.28** (*Arrays of Pointers to Functions*) Rewrite the program of Fig. 6.22 to use a menu driven interface. The program should offer the user 4 options as follows:

```
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. 6.22 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. 6.22 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.

**7.29** (*Modifications to the Simpletron Simulator*) In Exercise 7.19, 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 Exercises 12.26 and 12.27, we propose building a compiler that converts programs written in a high-level programming language (a variation of BASIC) to Simpletron Machine Language. Some of the following modifications and enhancements may be required to execute the programs produced by the compiler.

- 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.

**7.30** What does this program do?

---

```

1  /* ex07_30.c */
2  #include <stdio.h>
3
4  int mystery3( const char *, const char * );
5
6  int main()
7  {
8      char string1[ 80 ], string2[ 80 ];
9
10     printf( "Enter two strings: " );
11     scanf( "%s%s", string1 , string2 );
12     printf( " The result is %d\n",
13           mystery3( string1, string2 ) );
14
15     return 0;
16 }
17
18 int mystery3( const char *s1, const char *s2 )
19 {
20     for ( ; *s1 != '\0' && *s2 != '\0'; s1++, s2++ )
21         if ( *s1 != *s2 )
22             return 0;
23
24     return 1;
25 }
26

```

---