Documento de lecturas complementarias

La bibliografía relacionada a este material pueden encontrarse en
https://www.dropbox.com/s/nxp8bva28h9o0f8/prgrmmng_II.pdf?dl=0

28 de febrero de 2020

Parte I

Pointers (Deitel)

7

C Pointers

Objectives

In this chapter, you'll:

- Use pointers and pointer operators.
- Pass arguments to functions by reference using pointers.
- Understand the various placements of the `const` qualifier and how they affect what you can do with a variable.
- Use the `sizeof` operator with variables and types.
- Use pointer arithmetic to process the elements in arrays.
- Understand the close relationships among pointers, arrays and strings.
- Define and use arrays of strings.
- Use pointers to functions.
- Learn about secure C programming issues with regard to pointers.

7.1 Introduction

In this chapter, we discuss one of the most powerful features of the C programming language, the **pointer**.[1] Pointers are among C's most difficult capabilities to master. Pointers enable programs to accomplish pass-by-reference, to pass functions between functions, and to create and manipulate dynamic data structures—ones that can grow and shrink at execution time, such as linked lists, queues, stacks and trees. This chapter explains basic pointer concepts. In Section 7.13, we discuss various pointer-related security issues. 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.

1. Pointers and pointer-based entities such as arrays and strings, when misused intentionally or accidentally, can lead to errors and security breaches. See our Secure C Programming Resource Center (`www.deitel.com/SecureC/`) for articles, books, white papers and forums on this important topic.

7.2 Pointer Variable Definitions and Initialization

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

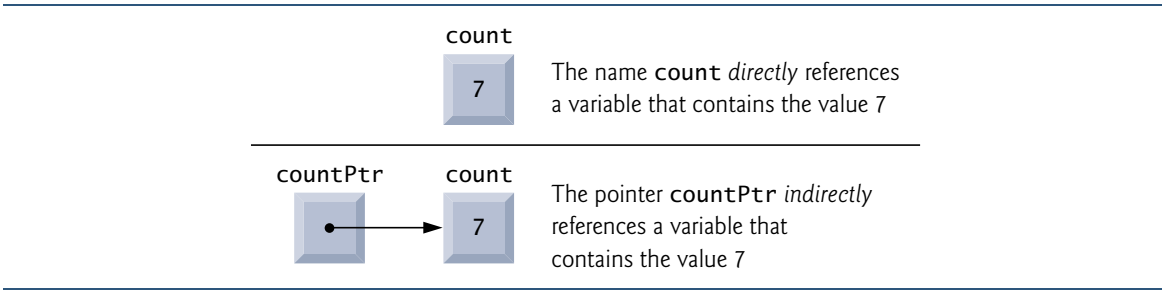


Fig. 7.1 | Directly and indirectly referencing a variable.

Declaring Pointers

Pointers, like all variables, must be defined before they can be used. The definition

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

specifies that variable `countPtr` is of type `int *` (i.e., a pointer to an integer) and is read (right to left), "`countPtr` is a pointer to `int`" or "`countPtr` points to an object[2] of type `int`." Also, the variable `count` is defined to be an `int`, *not* a pointer to an `int`. The `*` applies *only* to `countPtr` in the definition. When `*` is used in this manner in a definition, it indicates that the variable being defined is a pointer. Pointers can be defined to point to objects of any type. To prevent the ambiguity of declaring pointer and non-pointer variables in the same declaration as shown above, you should always declare only one variable per declaration.

Common Programming Error 7.1

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

Good Programming Practice 7.1

We prefer to include the letters `Ptr` *in pointer variable names to make it clear that these variables are pointers and need to be handled appropriately.*

Initializing and Assigning Values to Pointers

Pointers should be initialized when they're defined, or they can be assigned a value. A pointer may be initialized to `NULL`, `0` or an address. A pointer with the value `NULL` points to *nothing*. `NULL` is a *symbolic constant* defined in the `<stddef.h>` header (and several other headers, such as `<stdio.h>`). Initializing a pointer to `0` is equivalent to initializing a pointer to `NULL`, but `NULL` is preferred, because it highlights the fact that the variable is of a pointer

2. In C, an "object" is a region of memory that can hold a value. So objects in C include primitive types such as `int`s, `float`s, `char`s and `double`s, as well as aggregate types such as arrays and `struct`s (which we discuss in Chapter 10).

type. When 0 is assigned, it's first converted to a pointer of the appropriate type. The value 0 is the *only* integer value that can be assigned directly to a pointer variable. Assigning a variable's address to a pointer is discussed in Section 7.3.

Error-Prevention Tip 7.1

Initialize pointers to prevent unexpected results.

7.3 Pointer Operators

In this section, we present the address (&) and indirection (*) operators, and the relationship between them.

The Address (&) Operator

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

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

the statement

```
yPtr = &y;
```

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

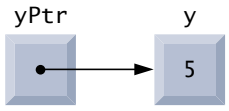


Fig. 7.2 | Graphical representation of a pointer pointing to an integer variable in memory.

Pointer Representation in Memory

Figure 7.3 shows the representation of the preceding pointer in memory, assuming that integer variable y is stored at location 600000, and pointer variable yPtr is stored at location 500000. The operand of the address operator must be a variable; the address operator *cannot* be applied to constants or expressions.

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

The Indirection (*) Operator

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

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

prints the value of variable y (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 is an error. This could cause a fatal execution-time error, or it could accidentally modify important data and allow the program to run to completion with incorrect results.

*Demonstrating the & and * Operators*

Figure 7.4 demonstrates the pointer operators & and *. The `printf` conversion specifier `%p` outputs the memory location as a *hexadecimal* integer on most platforms. (See Appendix C for more information on hexadecimal integers.) In the program's output, notice that the *address* of a and the *value* of `aPtr` are identical in the output, thus confirming that the address of a is indeed assigned to the pointer variable `aPtr` (line 8). The & and * operators are complements of one another—when they're both applied consecutively to `aPtr` in either order (line 18), the same result is printed. The addresses shown in the output will vary across systems. Figure 7.5 lists the precedence and associativity of the operators introduced to this point.

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

```
The address of a is 0028FEC0
The value of aPtr is 0028FEC0

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

Showing that * and & are complements of each other
&*aPtr = 0028FEC0
*&aPtr = 0028FEC0
```

Fig. 7.4 | Using the & and * pointer operators.

Operators	Associativity	Type
`()` `[]` `++` *(postfix)* `--` *(postfix)*	left to right	postfix
`+` `-` `++` `--` `!` `*` `&` `(`*type*`)`	right to left	unary
`*` `/` `%`	left to right	multiplicative
`+` `-`	left to right	additive
`<` `<=` `>` `s>=`	left to right	relational
`==` `!=`	left to right	equality
`&&`	left to right	logical AND
`||`	left to right	logical OR
`?:`	right to left	conditional
`=` `+=` `-=` `*=` `/=` `%=`	right to left	assignment
`,`	left to right	comma

Fig. 7.5 | Precedence and associativity of the operators discussed so far.

7.4 Passing Arguments to Functions by Reference

There are two ways to pass arguments to a function—**pass-by-value** and **pass-by-reference**. *However, all arguments in C are passed by value.* Functions often require the capability to *modify variables in the caller* or receive a pointer to a large data object to avoid the overhead of receiving the object by value (which incurs the time and memory overheads of making a copy of the object). 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). Pass-by-reference also can be used to enable a function to "return" multiple values to its caller by modifying variables in the caller.

*Use & and * to Accomplish Pass-By-Reference*

In C, you use pointers and the indirection operator to accomplish pass-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 (in the caller) whose value will be modified. As we saw in Chapter 6, arrays are *not* passed using operator `&` because C automatically passes the starting location in memory of the array (the name of an array is equivalent to `&arrayName[0]`). When the address of a variable is passed to a function, the indirection operator (`*`) may be used in the function to modify the value at that location in the caller's memory.

Pass-By-Value

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

```c
// Fig. 7.6: fig07_06.c
// Cube a variable using pass-by-value.
#include <stdio.h>

int cubeByValue(int n); // prototype

int main(void)
{
   int number = 5; // initialize number

   printf("The original value of number is %d", number);

   // pass number by value to cubeByValue
   number = cubeByValue(number);

   printf("\nThe new value of number is %d\n", number);
}

// calculate and return cube of integer argument
int cubeByValue(int n)
{
   return n * n * n; // cube local variable n and return result
}
```

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

Fig. 7.6 | Cube a variable using pass-by-value.

Pass-By-Reference

Figure 7.7 passes the variable `number` by reference (line 15)—the address of `number` is passed—to function `cubeByReference`. Function `cubeByReference` takes as a parameter a pointer to an `int` called `nPtr` (line 21). The function *dereferences* the pointer and cubes the value to which `nPtr` points (line 23), then assigns the result to `*nPtr` (which is really `number` in `main`), thus changing the value of `number` in `main`. Figures 7.8 and 7.9 analyze graphically and step-by-step the programs in Figs. 7.6 and 7.7, respectively.

```c
// Fig. 7.7: fig07_07.c
// Cube a variable using pass-by-reference with a pointer argument.

#include <stdio.h>

void cubeByReference(int *nPtr); // function prototype

int main(void)
{
```

Fig. 7.7 | Cube a variable using pass-by-reference with a pointer argument. (Part 1 of 2.)

```
10      int number = 5; // initialize number
11
12      printf("The original value of number is %d", number);
13
14      // pass address of number to cubeByReference
15      cubeByReference(&number);
16
17      printf("\nThe new value of number is %d\n", number);
18   }
19
20   // calculate cube of *nPtr; actually modifies number in main
21   void cubeByReference(int *nPtr)
22   {
23      *nPtr = *nPtr * *nPtr * *nPtr; // cube *nPtr
24   }
```

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

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

Use a Pointer Parameter to Receive an Address

A function receiving an *address* as an argument must define a *pointer parameter* to receive the address. For example, in Fig. 7.7 the header for function `cubeByReference` (line 21) 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.

Pointer Parameters in Function Prototypes

The function prototype for `cubeByReference` (Fig. 7.7, line 6) specifies an `int *` parameter. As with other variable types, it's *not* necessary to include names of pointers in function prototypes. Names included for documentation purposes are ignored by the C compiler.

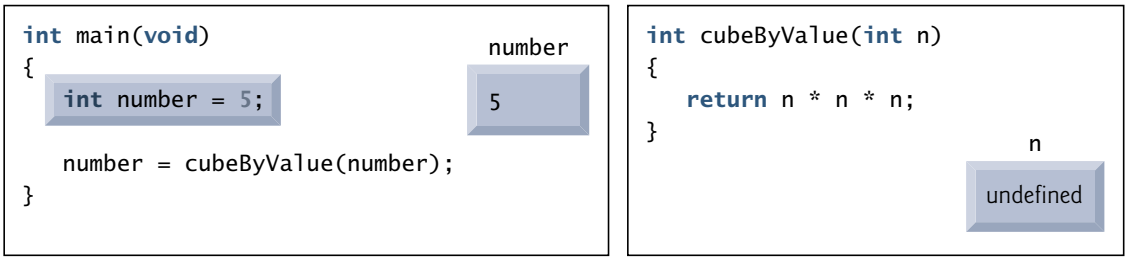
Functions That Receive One-Dimensional Arrays

For a function that expects a one-dimensional array as an argument, the function's prototype and header can use the pointer notation shown in the parameter list of function `cubeByReference` (line 21). The compiler does not differentiate between a function that receives a pointer and one that receives a one-dimensional array. This, of course, means that the function must "know" when it's receiving an array or simply a single variable for which it's to perform pass-by-reference. When the compiler encounters a function parameter for a one-dimensional array of the form `int b[]`, the compiler converts the parameter to the pointer notation `int *b`. The two forms are interchangeable.

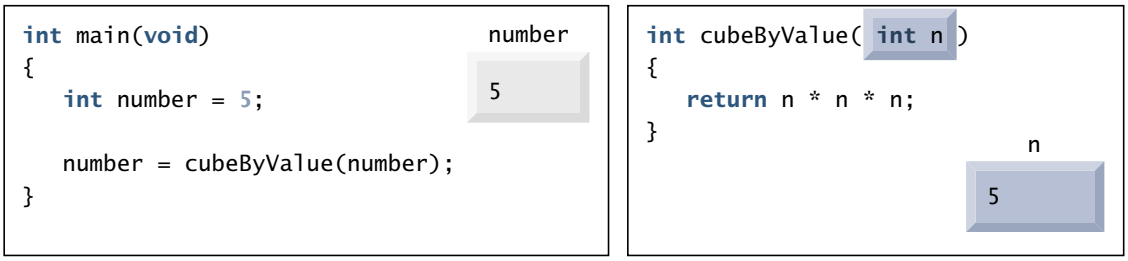
Error-Prevention Tip 7.2

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

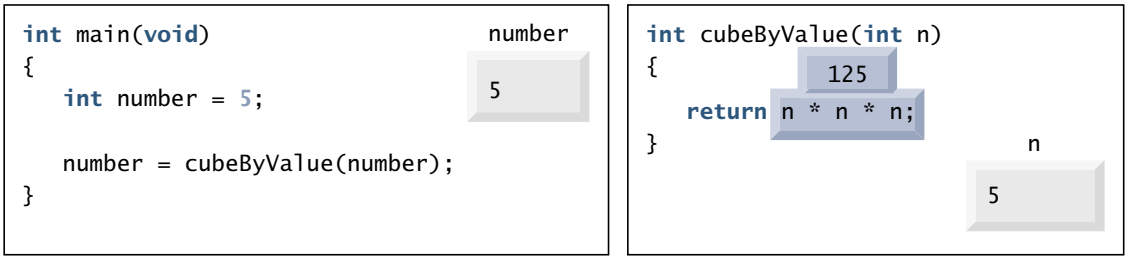
Step 1: Before `main` calls `cubeByValue`:



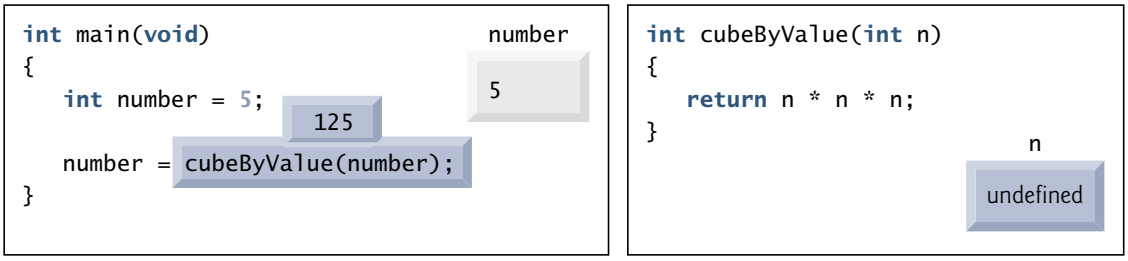
Step 2: After `cubeByValue` receives the call:



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



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



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

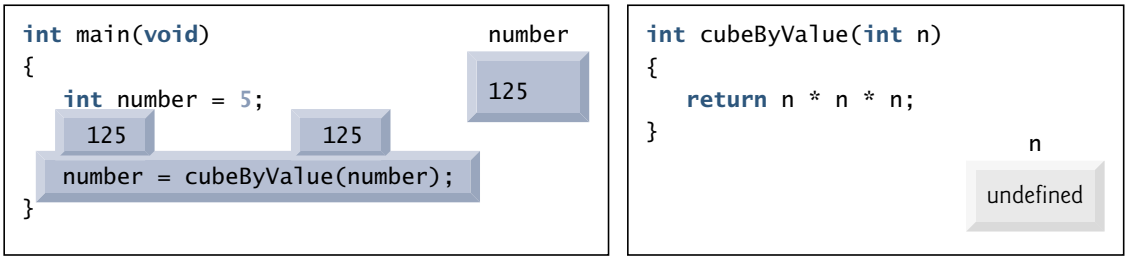


Fig. 7.8 | Analysis of a typical pass-by-value.

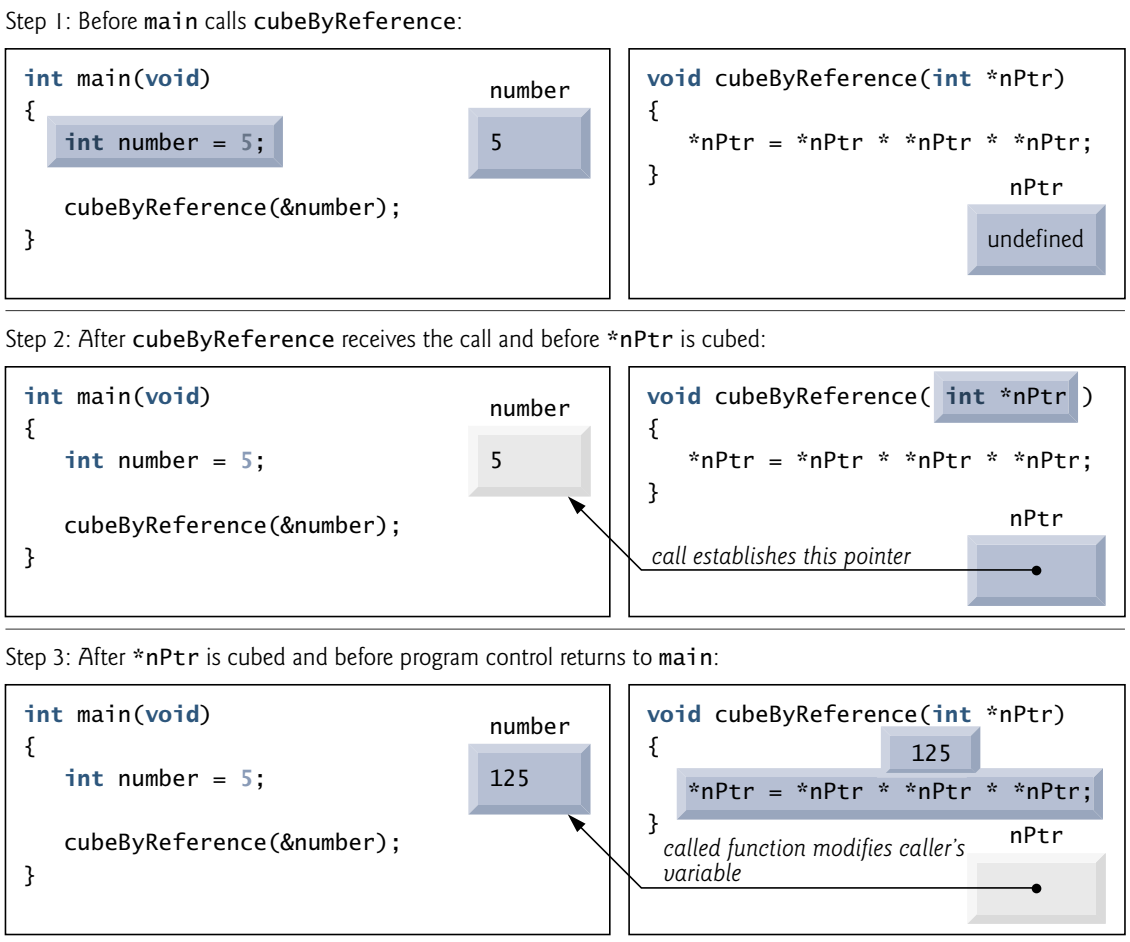


Fig. 7.9 | Analysis of a typical pass-by-reference with a pointer argument.

7.5 Using the const Qualifier with Pointers

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

Software Engineering Observation 7.1

The const qualifier can be used to enforce the principle of least privilege in software design. This can reduce debugging time and prevent unintentional side effects, making a program easier to modify and maintain.

Over the years, a large base of legacy code was written in early versions of C that did not use const because it was not available. For this reason, there are significant opportunities for improvement by reengineering old C code.

Six possibilities exist for using (or not using) const with function parameters—two with pass-by-value parameter passing and four with pass-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 absolutely no more.

`const` Values and Parameters

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

Consider a function that takes a one-dimensional 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 when the loop should terminate. Neither the size of the array nor its contents should change in the function body.

Error-Prevention Tip 7.3

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

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

Common Programming Error 7.3

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

There are four ways to pass a pointer to a function:

- a **non-constant pointer to non-constant data**.
- a **constant pointer to nonconstant data**.
- a **non-constant pointer to constant data**.
- a **constant pointer to constant data**.

Each of the four combinations provides different access privileges and is discussed in the next several examples.

7.5.1 Converting a String to Uppercase Using a Non-Constant Pointer to Non-Constant Data

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 processes (and possibly modifies) each character in the string. Function `convertToUppercase` of Fig. 7.10 declares its parameter, a *non-constant pointer to non-constant data* called `sPtr` (`char *sPtr`), in line 19. The function processes the array `string` (pointed to by `sPtr`) one character at a time. C standard library function

`toupper` (line 22) from the `<ctype.h>` header is called to convert each character to its corresponding uppercase letter—if the original character is not a letter or is already uppercase, `toupper` returns the original character. Line 23 moves the pointer to the next character in the string. Chapter 8 presents many C standard library character- and string-processing functions.

```
1  // Fig. 7.10: fig07_10.c
2  // Converting a string to uppercase using a
3  // non-constant pointer to non-constant data.
4  #include <stdio.h>
5  #include <ctype.h>
6
7  void convertToUppercase(char *sPtr); // prototype
8
9  int main(void)
10 {
11    char string[] = "cHaRaCters and $32.98"; // initialize char array
12
13    printf("The string before conversion is: %s", string);
14    convertToUppercase(string);
15    printf("\nThe string after conversion is: %s\n", string);
16 }
17
18 // convert string to uppercase letters
19 void convertToUppercase(char *sPtr)
20 {
21    while (*sPtr != '\0') { // current character is not '\0'
22       *sPtr = toupper(*sPtr); // convert to uppercase
23       ++sPtr; // make sPtr point to the next character
24    }
25 }
```

```
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.

7.5.2 Printing a String One Character at a Time Using a Non-Constant Pointer to Constant Data

A **non-constant pointer to constant data** *can be modified* to point to any data item of the appropriate type, but the *data* to which it points *cannot be modified*. Such a pointer might be used to receive an array argument to a function that will process each element without modifying that element. For example, function `printCharacters` (Fig. 7.11) declares parameter `sPtr` to be of type `const char *` (line 21). The declaration is read from *right to left* as "`sPtr` is a pointer to a character constant." The function uses a `for` statement to output each character in the string until the null character is encountered. After each character is printed, pointer `sPtr` is incremented—this makes the pointer move to the next character in the string.

```c
// Fig. 7.11: fig07_11.c
// Printing a string one character at a time using
// a non-constant pointer to constant data.

#include <stdio.h>

void printCharacters(const char *sPtr);

int main(void)
{
   // initialize char array
   char string[] = "print characters of a string";

   puts("The string is:");
   printCharacters(string);
   puts("");
}

// sPtr cannot be used to modify the character to which it points,
// i.e., sPtr is a "read-only" pointer
void printCharacters(const char *sPtr)
{
   // loop through entire string
   for (; *sPtr != '\0'; ++sPtr) { // no initialization
      printf("%c", *sPtr);
   }
}
```

```
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.

Figure 7.12 illustrates the attempt to compile a function that receives a non-constant pointer (`xPtr`) to constant data. This function attempts to modify the data pointed to by `xPtr` in line 18—which results in a compilation error. The error shown is from the Visual C++ compiler. The actual error message you receive (in this and other examples) is compiler specific—for example, Xcode's LLVM compiler reports the error:

```
Read-only variable is not assignable"
```

and the GNU gcc compiler reports the error:

```
error: assignment of read-only location '*xPtr'
```

```c
// Fig. 7.12: fig07_12.c
// Attempting to modify data through a
// non-constant pointer to constant data.
```

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

```
4  #include <stdio.h>
5  void f(const int *xPtr); // prototype
6
7  int main(void)
8  {
9     int y; // define y
10
11    f(&y); // f attempts illegal modification
12 }
13
14 // xPtr cannot be used to modify the
15 // value of the variable to which it points
16 void f(const int *xPtr)
17 {
18    *xPtr = 100; // error: cannot modify a const object
19 }
```

```
error C2166: l-value specifies const object
```

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

As you know, arrays are aggregate data types that store related data items of the same type under one name. In Chapter 10, we'll discuss another form of aggregate data type called a **structure** (sometimes called a **record** or tuple in other languages). A structure is capable of storing related data items of the same or *different* data types under one name (e.g., storing information about each employee of a company). When a function is called with an array as an argument, the array is automatically passed to the function *by reference*. However, structures are always passed *by value*—a *copy* of the entire structure is passed. 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 pass-by-reference and the protection of pass-by-value. When a pointer to a structure is passed, only a copy of the *address* at which the structure is stored must be made. On a machine with four-byte addresses, a copy of four bytes of memory is made rather than a copy of a possibly large structure.

Performance Tip 7.1

Passing large objects such as structures by using pointers to constant data obtains the performance benefits of pass-by-reference and the security of pass-by-value.

If memory is low and execution efficiency is a concern, use pointers. If memory is in abundance and efficiency is not a major concern, pass data by value to enforce the principle of least privilege. Remember that some systems do not enforce `const` well, so pass-by-value is still the best way to prevent data from being modified.

7.5.3 Attempting to Modify a Constant Pointer to Non-Constant Data

A **constant pointer to non-constant data** always points to the *same* memory location, and the data at that location *can be modified* through the pointer. This is the default for an array

name. An array name is a constant pointer to the beginning of the array. All data in the array can be accessed and changed by using the array name and array indexing. 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 index notation. Pointers that are declared `const` must be initialized when they're defined (if the pointer is a function parameter, it's initialized with a pointer that's passed to the function). Figure 7.13 attempts to modify a constant pointer. Pointer `ptr` is defined in line 12 to be of type `int * const`. The definition is read from *right to left* as "`ptr` is a constant pointer to an integer." The pointer is initialized (line 12) with the address of integer variable `x`. The program attempts to assign the address of `y` to `ptr` (line 15), but the compiler generates an error message.

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

```
c:\examples\ch07\fig07_13.c(15) : error C2166: l-value specifies const object
```

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

7.5.4 Attempting to Modify a Constant Pointer to 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 index notation and does *not* modify the array. Figure 7.14 defines pointer variable `ptr` (line 13) to be of type `const int *const`, which is read from *right to left* as "`ptr` is a constant pointer to an integer constant." The figure shows the error messages generated when an attempt is made to *modify* the *data* to which `ptr` points (line 16) and when an attempt is made to *modify* the *address* stored in the pointer variable (line 17).

```
1  // Fig. 7.14: fig07_14.c
2  // Attempting to modify a constant pointer to constant data.
3  #include <stdio.h>
4
```

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

```c
int main(void)
{
   int x = 5; // initialize x
   int y; // define y

   // ptr is a constant pointer to a constant integer. ptr always
   // points to the same location; the integer at that location
   // cannot be modified
   const int *const ptr = &x; // initialization is OK

   printf("%d\n", *ptr);
   *ptr = 7; // error: *ptr is const; cannot assign new value
   ptr = &y; // error: ptr is const; cannot assign new address
}
```

```
c:\examples\ch07\fig07_14.c(16) : error C2166: l-value specifies const object
c:\examples\ch07\fig07_14.c(17) : error C2166: l-value specifies const object
```

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

7.6 Bubble Sort[3] Using Pass-by-Reference

Let's improve the bubble sort program of Fig. 6.15 to use two functions—`bubbleSort` and `swap` (Fig. 7.15). Function `bubbleSort` sorts the array. It calls function `swap` (line 46) to exchange the array elements `array[j]` and `array[j + 1]`.

```c
// Fig. 7.15: fig07_15.c
// Putting values into an array, sorting the values into
// ascending order and printing the resulting array.
#include <stdio.h>
#define SIZE 10

void bubbleSort(int * const array, const size_t size); // prototype

int main(void)
{
   // initialize array a
   int a[SIZE] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };

   puts("Data items in original order");

   // loop through array a
   for (size_t i = 0; i < SIZE; ++i) {
      printf("%4d", a[i]);
   }

```

Fig. 7.15 | Putting values into an array, sorting the values into ascending order and printing the resulting array. (Part 1 of 2.)

3. In Chapter 12 and Appendix D, we investigate sorting schemes that yield better performance.

```
21      bubbleSort(a, SIZE); // sort the array
22
23      puts("\nData items in ascending order");
24
25      // loop through array a
26      for (size_t i = 0; i < SIZE; ++i) {
27         printf("%4d", a[i]);
28      }
29
30      puts("");
31   }
32
33   // sort an array of integers using bubble sort algorithm
34   void bubbleSort(int * const array, const size_t size)
35   {
36      void swap(int *element1Ptr, int *element2Ptr); // prototype
37
38      // loop to control passes
39      for (unsigned int pass = 0; pass < size - 1; ++pass) {
40
41         // loop to control comparisons during each pass
42         for (size_t j = 0; j < size - 1; ++j) {
43
44            // swap adjacent elements if they're out of order
45            if (array[j] > array[j + 1]) {
46               swap(&array[j], &array[j + 1]);
47            }
48         }
49      }
50   }
51
52   // swap values at memory locations to which element1Ptr and
53   // element2Ptr point
54   void swap(int *element1Ptr, int *element2Ptr)
55   {
56      int hold = *element1Ptr;
57      *element1Ptr = *element2Ptr;
58      *element2Ptr = hold;
59   }
```

```
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 | Putting values into an array, sorting the values into ascending order and printing the resulting array. (Part 2 of 2.)

Function `swap`

Remember that C enforces *information hiding* between functions, so `swap` does not have access to individual array elements in `bubbleSort` by default. Because `bubbleSort` *wants* `swap` to have access to the array elements to be swapped, `bubbleSort` passes each of these elements *by reference* to `swap`—the *address* of each array element is passed explicitly. Although entire

arrays are automatically passed by reference, individual array elements are *scalars* and are ordinarily passed by value. Therefore, `bubbleSort` uses the address operator (`&`) on each of the array elements in the `swap` call (line 46) to effect pass-by-reference as follows

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

Function `swap` receives `&array[j]` in `element1Ptr` (line 54). Even though `swap`—because of information hiding—is *not* allowed to know the name `array[j]`, `swap` may use `*element1Ptr` as a *synonym* for `array[j]`—when `swap` accesses `*element1Ptr`, it's *actually* referencing `array[j]` in `bubbleSort`. Similarly, when `swap` accesses `*element2Ptr`, it's *actually* referencing `array[j + 1]` in `bubbleSort`. Even though `swap` is not allowed to say

```
int hold = array[j];
array[j] = array[j + 1];
array[j + 1] = hold;
```

precisely the *same* effect is achieved by lines 56 through 58

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

Function `bubbleSort`*'s Array Parameter*

Several features of function `bubbleSort` should be noted. The function header (line 34) declares `array` as `int * const array` rather than `int array[]` to indicate that `bubbleSort` receives a one-dimensional array 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's *not* modified.

Function `swap`*'s Prototype in Function* `bubbleSort`*'s Body*

The prototype for function `swap` (line 36) is included in the body of function `bubbleSort` because `bubbleSort` is the only function that calls `swap`. Placing the prototype in `bubbleSort` restricts proper calls of `swap` to those made from `bubbleSort` (or any function that appears after `swap` in the source code). Other functions defined before `swap` that attempt to call `swap` do *not* have access to a proper function prototype, so the compiler generates one automatically. This normally results in a prototype that does *not* match the function header (and generates a compilation warning or error) because the compiler assumes `int` for the return type and the parameter types.

Software Engineering Observation 7.2

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.

Function `bubbleSort`*'s* `size` *Parameter*

Function `bubbleSort` receives the size of the array as a parameter (line 34). The function must know the size of the array to sort the array. When an array is passed to a function,

the memory address of the first element of the array is received by the function. The address, of course, does *not* convey the number of elements in the array. Therefore, you must pass the array size to the function. Another common practice is to pass a pointer to the beginning of the array and a pointer to the location just beyond the end of the array—as you'll learn in Section 7.8, the difference of the two pointers is the length of the array and the resulting code is simpler.

In the program, the size of the array is explicitly passed to function `bubbleSort`. There are two main benefits to this approach—*software reusability* and *proper software engineering*. By defining the function to receive the array size as an argument, we enable the function to be used by any program that sorts one-dimensional integer arrays of any size.

Software Engineering Observation 7.3

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

We could have stored the array's size in a global variable that's 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 7.4

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

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

7.7 `sizeof` Operator

C provides the special unary operator `sizeof` to determine the size in bytes of an array (or any other data type). This operator is applied at compilation time, unless its operand is a variable-length array (Section 6.12). When applied to the name of an array as in Fig. 7.16 (line 15), the `sizeof` operator returns the total number of bytes in the array as type `size_t`.[4] Variables of type `float` on this computer are stored in 4 bytes of memory, and `array` is defined to have 20 elements. Therefore, there are a total of 80 bytes in `array`.

Performance Tip 7.2

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

4. Recall that on a Mac `size_t` represents `unsigned long`. Xcode reports warnings when you display an `unsigned long` using `"%u"` in a `printf`. To eliminate the warnings, use `"%lu"` instead.

```c
// Fig. 7.16: fig07_16.c
// Applying sizeof to an array name returns
// the number of bytes in the array.
#include <stdio.h>
#define SIZE 20

size_t getSize(float *ptr); // prototype

int main(void)
{
   float array[SIZE]; // create array

   printf("The number of bytes in the array is %u"
          "\nThe number of bytes returned by getSize is %u\n",
          sizeof(array), getSize(array));
}

// return size of ptr
size_t getSize(float *ptr)
{
   return sizeof(ptr);
}
```

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

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

The number of elements in an array also can be determined with `sizeof`. For example, consider the following array definition:

```c
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:

```c
sizeof(real) / sizeof(real[0])
```

The expression determines the number of bytes in array `real` and divides that value by the number of bytes used in memory to store the first element of array `real` (a `double` value).

Even though function `getSize` receives an array of 20 elements as an argument, the function's parameter `ptr` is simply a pointer to the array's first element. When you use `sizeof` with a pointer, it returns the *size of the pointer*, not the size of the item to which it points. On our Windows and Linux test systems, the size of a pointer is 4 bytes, so `getSize` returns 4; on our Mac, the size of a pointer is 8 bytes, so `getSize` returns 8. Also, the calculation shown above for determining the number of array elements using `sizeof` works *only* when using the actual array, *not* when using a pointer to the array.

Determining the Sizes of the Standard Types, an Array and a Pointer

Figure 7.17 calculates the number of bytes used to store each of the standard data types. *The results of this program are implementation dependent and often differ across platforms and*

sometimes across different compilers on the same platform. The output shows the results from our Windows system using the Visual C++ compiler. The size of a `long double` was 12 bytes on our Linux system using the GNU gcc compiler. The size of a `long` was 8 bytes and the size of a `long double` was 16 bytes on our Mac system using Xcode's LLVM compiler.

```
1  // Fig. 7.17: fig07_17.c
2  // Using operator sizeof to determine standard data type sizes.
3  #include <stdio.h>
4
5  int main(void)
6  {
7     char c;
8     short s;
9     int i;
10    long l;
11    long long ll;
12    float f;
13    double d;
14    long double ld;
15    int array[20]; // create array of 20 int elements
16    int *ptr = array; // create pointer to array
17
18    printf("     sizeof c = %u\tsizeof(char)  = %u"
19           "\n     sizeof s = %u\tsizeof(short) = %u"
20           "\n     sizeof i = %u\tsizeof(int) = %u"
21           "\n     sizeof l = %u\tsizeof(long) = %u"
22           "\n    sizeof ll = %u\tsizeof(long long) = %u"
23           "\n     sizeof f = %u\tsizeof(float) = %u"
24           "\n     sizeof d = %u\tsizeof(double) = %u"
25           "\n    sizeof ld = %u\tsizeof(long double) = %u"
26           "\n sizeof array = %u"
27           "\n   sizeof ptr = %u\n",
28           sizeof c, sizeof(char), sizeof s, sizeof(short), sizeof i,
29           sizeof(int), sizeof l, sizeof(long), sizeof ll,
30           sizeof(long long), sizeof f, sizeof(float), sizeof d,
31           sizeof(double), sizeof ld, sizeof(long double),
32           sizeof array, sizeof ptr);
33 }
```

```
     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 ll = 8       sizeof(long long) = 8
     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.

Portability Tip 7.1

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.*

Operator `sizeof` can be applied to any variable name, type or value (including the value of an expression). When applied to a variable name (that's *not* an array name) or a constant, the number of bytes used to store the specific type of variable or constant is returned. The parentheses are required when a type is supplied as `sizeof`'s operand.

7.8 Pointer Expressions and Pointer Arithmetic

Pointers are valid operands in arithmetic expressions, assignment expressions and comparison expressions. However, not all the operators normally used in these expressions are valid in conjunction with pointer variables. This section describes the operators that can have pointers as operands, and how these operators are used.

7.8.1 Allowed Operators for Pointer Arithmetic

A pointer may be *incremented* (++) or *decremented* (--), an integer may be *added* to a pointer (+ or +=), an integer may be *subtracted* from a pointer (- or -=) and one pointer may be subtracted from another—this last operation is meaningful only when *both* pointers point to elements of the *same* array.

7.8.2 Aiming a Pointer at an Array

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

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

Portability Tip 7.2

Because the results of pointer arithmetic depend on the size of the objects a pointer points to, pointer arithmetic is machine and compiler dependent.

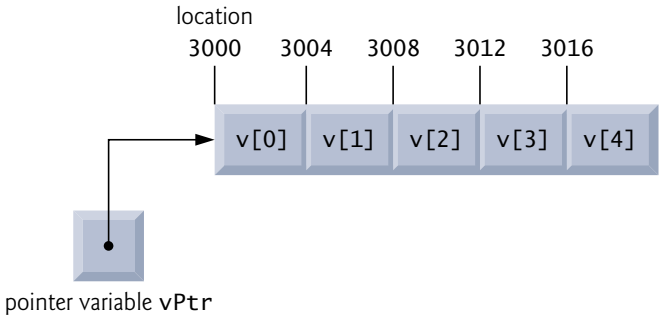


Fig. 7.18 | Array `v` and a pointer variable `vPtr` that points to `v`.

7.8.3 Adding an Integer to a Pointer

In conventional arithmetic, `3000 + 2` yields the value `3002`. This is normally *not* the case with pointer arithmetic. When an integer is added to or subtracted from a pointer, the pointer is *not* incremented or decremented simply by that integer, but by that integer times the size of the object to which the pointer refers. The number of bytes depends on the object's data type. For example, the statement

```
vPtr += 2;
```

would produce `3008` (`3000 + 2 * 4`), assuming an integer is stored in 4 bytes of memory. In the array `v`, `vPtr` would now point to `v[2]` (Fig. 7.19). If an integer is stored in 2 bytes of memory, then the preceding calculation would result in memory location `3004` (`3000 + 2 * 2`). If the array were of a different data type, the preceding statement would increment the pointer by twice the number of bytes that it takes to store an object of that data type. When performing pointer arithmetic on a character array, the results will be consistent with regular arithmetic, because each character is 1 byte long.

Common Programming Error 7.4

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

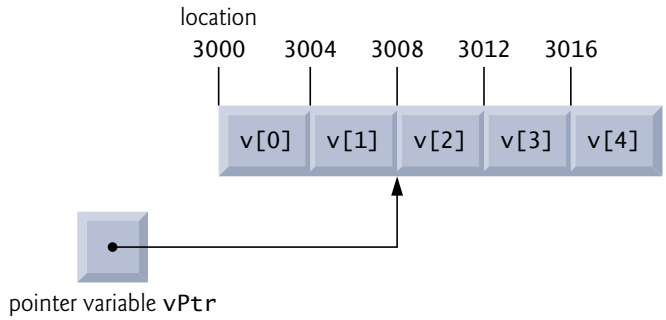


Fig. 7.19 | The pointer `vPtr` after pointer arithmetic.

7.8.4 Subtracting an Integer from a Pointer

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.

Common Programming Error 7.5

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

7.8.5 Incrementing and Decrementing a Pointer

If a pointer is being incremented or decremented by one, the increment (`++`) and decrement (`--`) operators can be used. Either of the statements

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

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

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

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

7.8.6 Subtracting One Pointer from Another

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 (not 8). Pointer arithmetic is undefined unless performed on an array. We cannot assume that two variables of the same type are stored contiguously in memory unless they're adjacent elements of an array.

Common Programming Error 7.6

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

7.8.7 Assigning Pointers to One Another

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

7.8.8 Pointer to `void`

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

Common Programming Error 7.7

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

Common Programming Error 7.8

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

7.8.9 Comparing Pointers

Pointers can be compared using equality and relational operators, but such comparisons are meaningless unless the pointers point to elements of the *same* array. Pointer comparisons compare the addresses stored in the pointers. A comparison of two pointers pointing to elements in the same array could show, for example, that one pointer points to a higher-

numbered element of the array than the other pointer does. A common use of pointer comparison is determining whether a pointer is `NULL`.

Common Programming Error 7.9

Comparing two pointers that do not refer to elements in the same *array.*

7.9 Relationship between Pointers and Arrays

Arrays and pointers are intimately related in C and often may be used interchangeably. An *array name* can be thought of as a *constant pointer*. Pointers can be used to do any operation involving array indexing.

Assume the following definitions:

```
int b[5];
int *bPtr;
```

Because the array name `b` (without an index) is a pointer to the array's first element, we can set `bPtr` equal to the address of the array `b`'s first element with the statement

```
bPtr = b;
```

This statement is equivalent to taking the address of array `b`'s first element as follows:

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

7.9.1 Pointer/Offset Notation

Array element `b[3]` can alternatively be referenced with the pointer expression

```
*(bPtr + 3)
```

The `3` in the expression is the **offset** to the pointer. When `bPtr` points to the array's first element, the offset indicates which array element to reference, and the offset value is identical to the array index. This 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 indexed array expressions can be written with a pointer and an offset. In this case, pointer/offset notation was used with the name of the array as a pointer. The preceding statement does not modify the array name in any way; `b` still points to the first element in the array.

7.9.2 Pointer/Index Notation

Pointers can be indexed like arrays. If `bPtr` has the value `b`, the expression

```
bPtr[1]
```

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

7.9.3 Cannot Modify an Array Name with Pointer Arithmetic

Remember that an array name always points to the beginning of the array—so the array name is like a constant pointer. Thus, the expression

```
b += 3
```

is *invalid* because it attempts to modify the array name's value with pointer arithmetic.

Common Programming Error 7.10

Attempting to modify the value of an array name with pointer arithmetic is a compilation error.

7.9.4 Demonstrating Pointer Indexing and Offsets

Figure 7.20 uses the four methods we've discussed for referring to array elements—array indexing, pointer/offset with the array name as a pointer, **pointer indexing**, 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 indexing and pointer notations with arrays.
3  #include <stdio.h>
4  #define ARRAY_SIZE 4
5
6  int main(void)
7  {
8     int b[] = {10, 20, 30, 40}; // create and initialize array b
9     int *bPtr = b; // create bPtr and point it to array b
10
11    // output array b using array index notation
12    puts("Array b printed with:\nArray index notation");
13
14    // loop through array b
15    for (size_t i = 0; i < ARRAY_SIZE; ++i) {
16       printf("b[%u] = %d\n", i, b[i]);
17    }
18
19    // output array b using array name and pointer/offset notation
20    puts("\nPointer/offset notation where\n"
21          "the pointer is the array name");
22
23    // loop through array b
24    for (size_t offset = 0; offset < ARRAY_SIZE; ++offset) {
25       printf("*(b + %u) = %d\n", offset, *(b + offset));
26    }
```

Fig. 7.20 | Using indexing and pointer notations with arrays. (Part 1 of 2.)

```
27
28      // output array b using bPtr and array index notation
29      puts("\nPointer index notation");
30
31      // loop through array b
32      for (size_t i = 0; i < ARRAY_SIZE; ++i) {
33         printf("bPtr[%u] = %d\n", i, bPtr[i]);
34      }
35
36      // output array b using bPtr and pointer/offset notation
37      puts("\nPointer/offset notation");
38
39      // loop through array b
40      for (size_t offset = 0; offset < ARRAY_SIZE; ++offset) {
41         printf("*(bPtr + %u) = %d\n", offset, *(bPtr + offset));
42      }
43   }
```

```
Array b printed with:
Array index 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 index 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 indexing and pointer notations with arrays. (Part 2 of 2.)

7.9.5 String Copying with Arrays and Pointers

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

```c
// Fig. 7.21: fig07_21.c
// Copying a string using array notation and pointer notation.
#include <stdio.h>
#define SIZE 10

void copy1(char * const s1, const char * const s2); // prototype
void copy2(char *s1, const char *s2); // prototype

int main(void)
{
   char string1[SIZE]; // create array string1
   char *string2 = "Hello"; // create a pointer to a string

   copy1(string1, string2);
   printf("string1 = %s\n", string1);

   char string3[SIZE]; // create array string3
   char string4[] = "Good Bye"; // create an array containing a string

   copy2(string3, string4);
   printf("string3 = %s\n", string3);
}

// copy s2 to s1 using array notation
void copy1(char * const s1, const char * const s2)
{
   // loop through strings
   for (size_t i = 0; (s1[i] = s2[i]) != '\0'; ++i) {
      ; // do nothing in body
   }
}

// copy s2 to s1 using pointer notation
void copy2(char *s1, const char *s2)
{
   // loop through strings
   for (; (*s1 = *s2) != '\0'; ++s1, ++s2) {
      ; // do nothing in body
   }
}
```

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

Fig. 7.21 | Copying a string using array notation and pointer notation.

Copying with Array Index Notation

Function `copy1` uses *array index notation* to copy the string in `s2` to the character array `s1`. The function defines counter variable `i` as the array index. The `for` statement header (line 28) performs the entire copy operation—its body is the empty statement. The header specifies that `i` is initialized to zero and incremented by one on each iteration of the loop. The expression `s1[i] = s2[i]` copies one character from `s2` to `s1`. When the null character

is encountered in `s2`, it's assigned to `s1`, and the value of the assignment becomes the value assigned to the left operand (`s1`). The loop terminates when the null character is assigned from `s2` to `s1` (false).

Copying with Pointers and Pointer Arithmetic

Function `copy2` uses *pointers and pointer arithmetic* to copy the string in `s2` to the character array `s1`. Again, the `for` statement header (line 37) performs the entire copy operation. The header does not include any variable initialization. As in function `copy1`, the expression (`*s1 = *s2`) performs the copy operation. Pointer `s2` is dereferenced, and the resulting character is assigned to the dereferenced pointer `*s1`. After the assignment in the condition, the pointers are incremented to point to the next character in the array `s1` and the next character in the string `s2`, respectively. When the null character is encountered in `s2`, it's assigned to the dereferenced pointer `s1` and the loop terminates.

Notes Regarding Functions **copy1** *and* **copy2**

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's not part of the array. Also, the second parameter of each function is declared as `const char * const` (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 that the *principle of least privilege* is enforced—neither function requires the capability of modifying the string in the second argument.

7.10 Arrays of Pointers

Arrays may contain pointers. A common use of an **array of pointers** is to form an **array of strings**, referred to simply as a **string array**. Each entry in the array is a string, but in C a string is essentially a pointer to its first character. So each entry in an array of strings is actually a pointer to the first character of a string. Consider the definition of string array `suit`, which might be useful in representing a deck of cards.

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

The `suit[4]` portion of the definition indicates an array of 4 elements. The `char *` portion of the declaration indicates that each element of array `suit` is of type "pointer to `char`." Qualifier `const` indicates that the strings pointed to by each element will not be modified. The four values to be placed in the array are `"Hearts"`, `"Diamonds"`, `"Clubs"` and `"Spades"`. Each is stored in memory as a *null-terminated character string* that's one character longer than the number of characters between the quotes. The four strings are 7, 9, 6 and 7 characters long, respectively. Although it appears 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 in a two-dimensional array, in which each row would represent a suit and each column would represent a letter from a suit name. Such a data structure would have to have a fixed number of columns per row, and that number

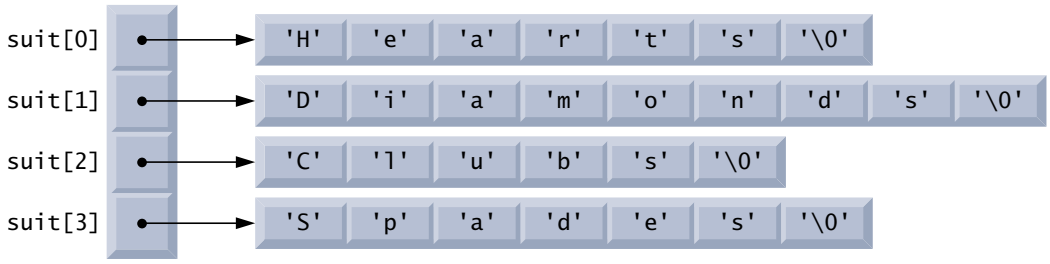


Fig. 7.22 | Graphical representation of the `suit` array.

would have to be as large as the largest string. Therefore, considerable memory could be wasted when storing a large number of strings of which most were shorter than the longest string. We use string arrays to represent a deck of cards in the next section.

7.11 Case Study: Card Shuffling and Dealing Simulation

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

Using the top-down, stepwise refinement approach, we develop a program that will shuffle a deck of 52 playing cards and then deal each of the 52 cards. The top-down approach is particularly useful in attacking larger, more complex problems than you've seen in earlier chapters.

Representing a Deck of Cards as a Two-Dimensional Array

We use 4-by-13 two-dimensional 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 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.

Shuffling the Two-Dimensional Array

This simulated deck of cards may be *shuffled* as follows. First the array `deck` is cleared to zeros. Then, a `row` (0–3) and a `column` (0–12) are each chosen *at random*. The number 1 is inserted in array element `deck[row][column]` to indicate that this card will 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's possible that a card will be selected again—i.e., `deck[row][column]` will be nonzero when it's selected. This selection is simply ignored and other `row`s and `column`s are repeatedly chosen at random until an *unselected* card is found. Eventually, the numbers 1 through 52 will occupy the 52 slots of the `deck` array. At this point, the deck of cards is fully shuffled.

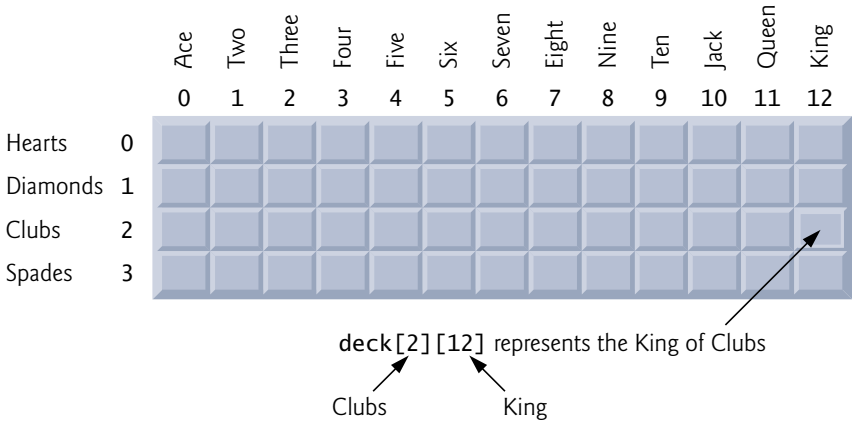


Fig. 7.23 | Two-dimensional array representation of a deck of cards.

Possibility of Indefinite Postponement

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

Performance Tip 7.3

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

Dealing Cards from the Two-Dimensional Array

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

Developing the Program's Logic with Top-Down, Stepwise Refinement

Let's proceed with the top-down, stepwise refinement process. The *top* is simply

Shuffle and deal 52 cards

Our *first refinement* yields:

Initialize the suit array
Initialize the face array
Initialize the deck array
Shuffle the deck
Deal 52 cards

"Shuffle the deck" may be expanded as follows:

```
For each of the 52 cards
    Place card number in randomly selected unoccupied element 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:

```
Choose slot of deck randomly

While chosen slot of deck has been previously chosen
    Choose slot of deck randomly

Place card number in chosen slot of deck
```

"Find card number in deck array and print face and suit of card" may be expanded as:

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

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. This program is more efficient if the shuffle and deal portions of the algorithm are combined so that each card is dealt as it's placed in

the deck. We've chosen to program these operations separately because normally cards are dealt after they're shuffled (not while they're being shuffled).

Implementing the Card Shuffling and Dealing Program

The card shuffling and dealing program is shown in Fig. 7.24, and a sample execution is shown in Fig. 7.25. Conversion specifier `%s` is used to print strings of characters in the calls to `printf`. The corresponding argument in the `printf` call must be a pointer to `char` (or a `char` array). The format specification `"%5s of %-8s"` (line 68) prints a character string *right justified* in a field of five characters followed by `" of "` and a character string *left justified* in a field of eight characters. The *minus sign* in `%-8s` signifies left justification.

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

```
1  // Fig. 7.24: fig07_24.c
2  // Card shuffling and dealing.
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <time.h>
6
7  #define SUITS 4
8  #define FACES 13
9  #define CARDS 52
10
11 // prototypes
12 void shuffle(unsigned int wDeck[][FACES]); // shuffling modifies wDeck
13 void deal(unsigned int wDeck[][FACES], const char *wFace[],
14    const char *wSuit[]); // dealing doesn't modify the arrays
15
16 int main(void)
17 {
18    // initialize deck array
19    unsigned int deck[SUITS][FACES] = {0};
20
21    srand(time(NULL)); // seed random-number generator
22    shuffle(deck); // shuffle the deck
23
24    // initialize suit array
25    const char *suit[SUITS] =
26       {"Hearts", "Diamonds", "Clubs", "Spades"};
27
28    // initialize face array
29    const char *face[FACES] =
30       {"Ace", "Deuce", "Three", "Four",
31        "Five", "Six", "Seven", "Eight",
32        "Nine", "Ten", "Jack", "Queen", "King"};
33
34    deal(deck, face, suit); // deal the deck
35 }
36
```

Fig. 7.24 | Card shuffling and dealing. (Part 1 of 2.)

```
37 // shuffle cards in deck
38 void shuffle(unsigned int wDeck[][FACES])
39 {
40    // for each of the cards, choose slot of deck randomly
41    for (size_t card = 1; card <= CARDS; ++card) {
42       size_t row; // row number
43       size_t column; // column number
44
45       // choose new random location until unoccupied slot found
46       do {
47          row = rand() % SUITS;
48          column = rand() % FACES;
49       } while(wDeck[row][column] != 0);
50
51       // place card number in chosen slot of deck
52       wDeck[row][column] = card;
53    }
54 }
55
56 // deal cards in deck
57 void deal(unsigned int wDeck[][FACES], const char *wFace[],
58    const char *wSuit[])
59 {
60    // deal each of the cards
61    for (size_t card = 1; card <= CARDS; ++card) {
62       // loop through rows of wDeck
63       for (size_t row = 0; row < SUITS; ++row) {
64          // loop through columns of wDeck for current row
65          for (size_t column = 0; column < FACES; ++column) {
66             // if slot contains current card, display card
67             if (wDeck[row][column] == card) {
68                printf("%5s of %-8s%c", wFace[column], wSuit[row],
69                   card % 2 == 0 ? '\n' : '\t'); // 2-column format
70             }
71          }
72       }
73    }
74 }
```

Fig. 7.24 | Card shuffling and dealing. (Part 2 of 2.)

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

Fig. 7.25 | Sample run of card dealing program. (Part 1 of 2.)

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

Fig. 7.25 | Sample run of card dealing program. (Part 2 of 2.)

7.12 Pointers to Functions

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

7.12.1 Sorting in Ascending or Descending Order

To illustrate the use of pointers to functions, Fig. 7.26 presents a modified version of the bubble sort program in Fig. 7.15. The new version consists of `main` and functions `bubble`, `swap`, `ascending` and `descending`. Function `bubbleSort` receives a pointer to a function—either function `ascending` or function `descending`—as an *argument*, in addition to an integer array and the size of the array. The program prompts the user to choose whether the array should be sorted in *ascending* or in *descending* order. If the user enters `1`, a pointer to function `ascending` is passed to function `bubble`, causing the array to be sorted into *increasing* order. If the user enters `2`, a pointer to function `descending` is passed to function `bubble`, causing the array to be sorted into *decreasing* order. The output of the program is shown in Fig. 7.27.

```
1  // Fig. 7.26: fig07_26.c
2  // Multipurpose sorting program using function pointers.
3  #include <stdio.h>
4  #define SIZE 10
5
```

Fig. 7.26 | Multipurpose sorting program using function pointers. (Part 1 of 3.)

```
6  // prototypes
7  void bubble(int work[], size_t size, int (*compare)(int a, int b) );
8  int ascending(int a, int b);
9  int descending(int a, int b);
10
11 int main(void)
12 {
13    // initialize unordered array a
14    int a[SIZE] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
15
16    printf("%s", "Enter 1 to sort in ascending order,\n"
17           "Enter 2 to sort in descending order: ");
18    int order; // 1 for ascending order or 2 for descending order
19    scanf("%d", &order);
20
21    puts("\nData items in original order");
22
23    // output original array
24    for (size_t counter = 0; counter < SIZE; ++counter) {
25       printf("%5d", a[counter]);
26    }
27
28    // sort array in ascending order; pass function ascending as an
29    // argument to specify ascending sorting order
30    if (order == 1) {
31       bubble(a, SIZE, ascending);
32       puts("\nData items in ascending order");
33    }
34    else { // pass function descending
35       bubble(a, SIZE, descending);
36       puts("\nData items in descending order");
37    }
38
39    // output sorted array
40    for (size_t counter = 0; counter < SIZE; ++counter) {
41       printf("%5d", a[counter]);
42    }
43
44    puts("\n");
45 }
46
47 // multipurpose bubble sort; parameter compare is a pointer to
48 // the comparison function that determines sorting order
49 void bubble(int work[], size_t size, int (*compare)(int a, int b))
50 {
51    void swap(int *element1Ptr, int *element2ptr); // prototype
52
53    // loop to control passes
54    for (unsigned int pass = 1; pass < size; ++pass) {
55
56       // loop to control number of comparisons per pass
57       for (size_t count = 0; count < size - 1; ++count) {
58
```

Fig. 7.26 | Multipurpose sorting program using function pointers. (Part 2 of 3.)

```
59            // if adjacent elements are out of order, swap them
60            if ((*compare)(work[count], work[count + 1])) {
61               swap(&work[count], &work[count + 1]);
62            }
63         }
64      }
65   }
66
67   // swap values at memory locations to which element1Ptr and
68   // element2Ptr point
69   void swap(int *element1Ptr, int *element2Ptr)
70   {
71      int hold = *element1Ptr;
72      *element1Ptr = *element2Ptr;
73      *element2Ptr = hold;
74   }
75
76   // determine whether elements are out of order for an ascending
77   // order sort
78   int ascending(int a, int b)
79   {
80      return b < a; // should swap if b is less than a
81   }
82
83   // determine whether elements are out of order for a descending
84   // order sort
85   int descending(int a, int b)
86   {
87      return b > a; // should swap if b is greater than a
88   }
```

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

```
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.

The following parameter appears in the function header for `bubble` (line 49)

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

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

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

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

The function prototype for `bubble` is shown in line 7. The third parameter in the prototype could have been written as

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

without the function-pointer name and parameter names.

The function passed to `bubble` is called in an `if` statement (line 60) 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's dereferenced to call the function. The second method of calling a function through a pointer makes it appear as if `compare` is an *actual* function. This may be confusing to a programmer reading the code who would like to see the definition of function `compare` and finds that it's *never defined* in the file.

7.12.2 Using Function Pointers to Create a Menu-Driven System

A common use of **function pointers** is in text-based *menu-driven systems*. A user is prompted to select an option from a menu (possibly from 1 to 5) by typing the menu item's number. Each option is serviced by a different function. Pointers to each function are stored in an array of pointers to functions. The user's choice is used as a index in 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 defining and using an array of pointers to functions. We define three functions—`function1`, `function2` and `function3`—that each take an integer argument and return nothing. We store pointers to these three functions in array `f`, which is defined in line 14. The definition is read beginning at the leftmost set of parentheses, "`f` is an array of `3` pointers to functions that each take an `int` as an argument and return `void`." The array is initialized with the names of the three functions. When the user enters a value between 0 and 2, the value is used as the index into the array of pointers to functions. In the function call (line 25), `f[choice]` selects the pointer at location `choice` in the array. The *pointer is dereferenced to call the func-*

tion, 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 this chapter's exercises, you'll develop several text-based, menu-driven systems.

```c
// Fig. 7.28: fig07_28.c
// Demonstrating an array of pointers to functions.
#include <stdio.h>

// prototypes
void function1(int a);
void function2(int b);
void function3(int c);

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

   printf("%s", "Enter a number between 0 and 2, 3 to end: ");
   size_t choice; // variable to hold user's choice
   scanf("%u", &choice);

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

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

      printf("%s", "Enter a number between 0 and 2, 3 to end: ");
      scanf("%u", &choice);
   }

   puts("Program execution completed.");
}

void function1(int a)
{
   printf("You entered %d so function1 was called\n\n", a);
}

void function2(int b)
{
   printf("You entered %d so function2 was called\n\n", b);
}

void function3(int c)
{
   printf("You entered %d so function3 was called\n\n", c);
}
```

Fig. 7.28 | Demonstrating an array of pointers to functions. (Part 1 of 2.)

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

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

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

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

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

7.13 Secure C Programming

`printf_s`, `scanf_s` and Other Secure Functions

Earlier Secure C Programming sections presented `printf_s` and `scanf_s`, and mentioned other more secure versions of standard library functions that are described by Annex K of the C standard. A key feature of functions like `printf_s` and `scanf_s` that makes them more secure is that they have *runtime constraints* requiring their pointer arguments to be non-`NULL`. The functions check these runtime constraints *before* attempting to use the pointers. Any `NULL` pointer argument is considered to be a *constraint violation* and causes the function to fail and return a status notification. In a `scanf_s`, if any of the pointer arguments (including the format-control string) are `NULL`, the function returns `EOF`. In a `printf_s`, if the format-control string or any argument that corresponds to a `%s` is `NULL`, the function stops outputting data and returns a negative number. For complete details of the Annex K functions, see the C standard document or your compiler's library documentation.

Other CERT Guidelines Regarding Pointers

Misused pointers lead to many of the most common security vulnerabilities in systems today. CERT provides various guidelines to help you prevent such problems. If you're building industrial-strength C systems, you should familiarize yourself with the *CERT C Secure Coding Standard* at `www.securecoding.cert.org`. The following guidelines apply to pointer programming techniques that we presented in this chapter:

- EXP34-C: Dereferencing `NULL` pointers typically causes programs to crash, but CERT has encountered cases in which dereferencing `NULL` pointers can allow attackers to execute code.
- DCL13-C: Section 7.5 discussed uses of `const` with pointers. If a function parameter points to a value that will not be changed by the function, `const` should be used to indicate that the data is constant. For example, to represent a pointer to a string that will not be modified, use `const char *` as the pointer parameter's type, as in line 21 of Fig. 7.11.
- MSC16-C: This guideline discusses techniques for encrypting function pointers to help prevent attackers from overwriting them and executing attack code.

Summary

Section 7.2 Pointer Variable Definitions and Initialization

- A **pointer** (p. 307) contains an address of another variable that contains a value. In this sense, a variable name *directly* references a value, and a pointer ***indirectly*** **references** a value.
- Referencing a value through a pointer is called **indirection** (p. 308).
- Pointers can be defined to point to objects of any type.
- Pointers should be initialized either when they're defined or in an assignment statement. A pointer may be initialized to `NULL`, `0` or **an address**. A pointer with the value `NULL` points to nothing. Initializing a pointer to `0` is equivalent to initializing a pointer to `NULL`, but `NULL` is preferred for clarity. The value `0` is the only integer value that can be assigned directly to a pointer variable.
- `NULL` is a symbolic constant defined in the `<stddef.h>` header (and several other headers).

Section 7.3 Pointer Operators

- The `&`, or **address operator** (p. 309), is a unary operator that returns the address of its operand.
- The operand of the address operator must be a variable.
- The **indirection operator** `*` (p. 309) returns the value of the object to which its operand points.
- The `printf` **conversion specifier** `%p` outputs a memory location as a hexadecimal integer on most platforms.

Section 7.4 Passing Arguments to Functions by Reference

- All arguments in C are **passed by value** (p. 311).
- C programs accomplish **pass-by-reference** (p. 311) by using pointers and the indirection operator. To pass a variable by reference, apply the address operator (`&`) to the variable's name.
- When the address of a variable is passed to a function, the indirection operator (`*`) may be used in the function to read and/or modify the value at that location in the caller's memory.
- A function receiving an address as an argument must define a **pointer parameter** to receive the address.
- The compiler does not differentiate between a function that receives a pointer and one that receives a **one-dimensional array**. A function must "know" when it's receiving an array vs. a single variable passed by reference.
- When the compiler encounters a function parameter for a one-dimensional array of the form `int b[]`, the compiler converts the parameter to the pointer notation `int *b`.

Section 7.5 Using the `const` *Qualifier with Pointers*

- The `const` **qualifier** (p. 315) indicates that the value of a particular variable should not be modified.
- If an attempt is made to modify a value that's 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 (p. 316): 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**.
- With a non-constant pointer to non-constant data, the data can be modified through the dereferenced pointer, and the pointer can be modified to point to other data items.
- A non-constant pointer to constant data can be modified to point to any data item of the appropriate type, but the data to which it points cannot be modified.
- A constant pointer to non-constant data always points to the same memory location, and the data at that location can be modified through the pointer. This is the default for an array name.

- A constant pointer to constant data always points to the same memory location, and the data at that memory location cannot be modified.

Section 7.7 `sizeof` Operator

- Unary **operator `sizeof`** (p. 324) determine the size in bytes of a variable or type at compilation time.
- When applied to the name of an array, `sizeof` returns the total number of bytes in the array.
- Operator `sizeof` can be applied to any variable name, type or value.
- The parentheses used with `sizeof` are required if a type name is supplied as its operand.

Section 7.8 Pointer Expressions and Pointer Arithmetic

- A limited set of **arithmetic operations** (p. 328) **may be performed on pointers**. A pointer may be **incremented** (`++`) or **decremented** (`--`), an integer may be **added** to a pointer (`+` or `+=`), an integer may be **subtracted** from a pointer (`-` or `-=`) and **one pointer may be subtracted from another**.
- When an integer is added to or subtracted from a pointer, the pointer is incremented or decremented by that integer times the size of the object to which the pointer refers.
- Two pointers to elements of the same array may be subtracted from one another to determine the number of elements between them.
- A pointer can be assigned to another pointer if both have the same type. An exception is the pointer of type `void *` (p. 329) which can represent any pointer type. All pointer types can be assigned a `void *` pointer, and a `void *` pointer can be assigned a pointer of any type.
- A `void *` pointer cannot be dereferenced.
- **Pointers can be compared** using equality and relational operators, but such comparisons are meaningless unless the pointers point to elements of the same array. Pointer comparisons compare the addresses stored in the pointers.
- A common use of pointer comparison is **determining whether a pointer is `NULL`**.

Section 7.9 Relationship between Pointers and Arrays

- Arrays and pointers are intimately related in C and often may be used interchangeably.
- An **array name** can be thought of as a **constant pointer**.
- Pointers can be used to do any operation involving array indexing.
- When a pointer points to the beginning of an array, adding an **offset** (p. 330) to the pointer indicates which element of the array should be referenced, and the offset value is identical to the array index. This is referred to as pointer/offset notation.
- An **array name can be treated as a pointer** and used in pointer arithmetic expressions that do not attempt to modify the address of the pointer.
- **Pointers can be indexed** (p. 331) exactly as arrays can. This is referred to as pointer/index notation.
- A parameter of type `const char *` typically represents a constant string.

Section 7.10 Arrays of Pointers

- **Arrays may contain pointers** (p. 334). A common use of an array of pointers is to form an **array of strings** (p. 334). 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.

Section 7.12 Pointers to Functions

- A **function pointer** (p. 343) contains the address of the function in memory. 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**.
- A pointer to a function is dereferenced to call the function. A function pointer can be used directly as the function name when calling the function.
- A common use of function pointers is in text-based, **menu-driven systems**.

Self-Review Exercises

7.1 Answer each of the following:

a) A pointer variable contains as its value the _____ of another variable.

b) The three values that can be used to initialize a pointer are _____, _____ and _____.

c) The only integer that can be assigned to a pointer is _____.

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

a) A pointer that's declared to be `void` can be dereferenced.

b) 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) Define 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) Define a pointer, `nPtr`, that points to an object of type `float`.

c) Print the elements of array `numbers` using array index notation. Use a `for` statement. 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 indexing pointer `nPtr`.

h) Refer to element 4 of array `numbers` using array index notation, pointer/offset notation with the array name as the pointer, pointer index 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's the value stored at that location?

7.4 For each of the following, write a statement that performs the indicated task. Assume that floating-point variables `number1` and `number2` are defined and that `number1` is initialized to `7.3`.

a) Define 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;
int z[5] = {1, 2, 3, 4, 5};
sPtr = z;
```

a)
```
++zptr;
```
b)
```
// use pointer to get first value of array; assume zPtr is initialized
number = zPtr;
```
c)
```
// assign array element 2 (the value 3) to number;
   assume zPtr is initialized
number = *zPtr[2];
```
d)
```
// print entire array z; assume zPtr is initialized
for (size_t 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. A pointer to `void` cannot be dereferenced, because there's no way to know exactly how many bytes of memory to dereference. b) 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 (size_t i = 0; i < SIZE; ++i) {
   printf("%.1f ", numbers[i]);
}
```
d)
```
nPtr = numbers;
nPtr = &numbers[0];
```
e)
```
for (size_t i = 0; i < SIZE; ++i) {
   printf("%.1f ", *(nPtr + i));
}
```

f)
```
for (size_t i = 0; i < SIZE; ++i) {
   printf("%.1f ", *(numbers + i));
}
```
g)
```
for (size_t i = 0; i < SIZE; ++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 *x, float *y);
```
c)
```
int evaluate(int x, int (*poly)(int))
```
d)
```
int evaluate(int x, int (*poly)(int));
```

7.6 a) Error: `zPtr` has not been initialized.
Correction: Initialize `zPtr` with `zPtr = z;` before performing the pointer arithmetic.
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 indexing.
Correction: Change the operator `<=` in the `for` condition to `<`.
e) Error: Dereferencing a void pointer.
Correction: To dereference the pointer, it must first be cast to an integer pointer. Change the 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 index the array name to refer to a specific element.

Exercises

7.7 Answer each of the following:
a) The _____ pointers are generic pointers that can hold pointer values of any type.
b) An array name can be thought of as a _____ pointer.
c) The _____ operator is a unary operator that returns the address of its operand, and the _____ operator returns the value of the object to which its operand points.

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

a) Arithmetic operations cannot be performed on pointers.
b) Pointers to functions can be used to call the functions they point to, passed to functions, returned from functions, stored in arrays, and assigned to other pointers.

7.9 Answer each of the following. Assume that integers are stored in 4 bytes and that the starting address of the array is at location 2003800 in memory.

a) Define an array of type `int` called `oddNum` with ten elements, and assign the odd integers from `1` to `19` to the elements. Assume the symbolic constant `SIZE` has been defined as `10`.
b) Define a pointer `iPtr` that points to an object of type `int`.
c) Print the elements of array `oddNum` using array index notation. Use a `for` statement and assume integer control variable `i` has been defined.
d) Give two separate statements that assign the starting address of array `oddNum` to pointer variable `iPtr`.
e) Print the elements of array `oddNum` using pointer/offset notation
f) Print the elements of array `oddNum` using pointer/offset notation with the array name as the pointer.
g) Print the elements of array `oddNum` by indexing the pointer to the array
h) Refer to element 3 of array `oddNum` using array index notation, pointer/offset notation with the array name as the pointer, pointer index notation, and pointer/offset notation.
i) What address is referenced by `iPtr + 5`? What value is stored at that location?
j) Assuming `iPtr` points to `oddNum[9]`, what address is referenced by `iPtr -= 3`? What value is stored at that location?

7.10 For each of the following, write a single statement that performs the specified task. Assume that double precision variables `value1` and `value2` have been declared and `value1` has been initialized to 20.4568.

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

7.11 Do each of the following.

a) Write the function header for function `addNumbers` that takes two values, a long integer array parameter `numList` and the size of the array, and returns the sum of the numbers.
b) Write the function prototype for the function in part (a).
c) Write the function header for a function `sort` that takes three arguments: an integer array parameter `n`, a constant integer size and a pointer to a function *f* that takes two integers and returns an integer value; sort does not return anything.
d) Write the function prototype for the function described in part (c).

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

7.12 *(Card Shuffling and Dealing)* 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 whether the hand contains a pair.
b) Determine whether the hand contains two pairs.

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

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

7.14 ***(Project: Card Shuffling and Dealing)*** 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 ***(Project: Card Shuffling and Dealing)*** 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 ***(Card Shuffling and Dealing Modification)*** 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'll 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 whether 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.

Unshuffled deck array

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

Fig. 7.29 | Unshuffled deck array.

Sample shuffled deck array

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

Fig. 7.30 | Sample shuffled `deck` array.

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 still searches 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'll recreate one of the truly great moments in history, namely the classic race of the tortoise and the hare. You'll use random number generation to develop a simulation of this memorable event.

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

There's a clock that ticks once per second. With each tick of the clock, your program should adjust the position of the animals according to the rules of Fig. 7.31.

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

Fig. 7.31 | Tortoise and hare rules for adjusting positions.

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 iteration 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 whether 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're ready to run your program, assemble a group of fans to watch the race. You'll be amazed at how involved your audience gets!

7.18 *(Card Shuffling and Dealing Modification)* 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's similar to function `shuffle` in Fig. 7.24.

7.19 What does this program do, assuming that the user enters two strings of the same length?

```
1  // ex07_19.c
2  // What does this program do?
3  #include <stdio.h>
4  #define SIZE 80
5
6  void mystery1(char *s1, const char *s2); // prototype
7
8  int main(void)
9  {
10    char string1[SIZE]; // create char array
11    char string2[SIZE]; // create char array
12
13    puts("Enter two strings: ");
14    scanf("%79s%79s" , string1, string2);
15    mystery1(string1, string2);
16    printf("%s", string1);
17 }
18
19 // What does this function do?
20 void mystery1(char *s1, const char *s2)
21 {
22    while (*s1 != '\0') {
23       ++s1;
24    }
25
26    for (; *s1 = *s2; ++s1, ++s2) {
27       ; // empty statement
28    }
29 }
```

7.20 What does this program do?

```
// ex07_20.c
// what does this program do?
#include <stdio.h>
#define SIZE 80

size_t mystery2(const char *s); // prototype

int main(void)
{
   char string[SIZE]; // create char array

   puts("Enter a string: ");
   scanf("%79s", string);
   printf("%d\n", mystery2(string));
}

// What does this function do?
size_t mystery2(const char *s)
{
   size_t x;

   // loop through string
   for (x = 0; *s != '\0'; ++s) {
      ++x;
   }

   return x;
}
```

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

a)
```
int *number;
printf("%d\n", *number);
```

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

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

d)
```
char s[] = "this is a character array";
int count;
for (; *s != '\0'; ++s)
   printf("%c ", *s);
```

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

f)
```
float x = 19.34;
float xPtr = &x;
printf("%f\n", xPtr);
```

g)
```
char *s;
printf("%s\n", s);
```

7.22 *(Maze Traversal)* The following grid is a two-dimensional array representation of a maze.

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

The # symbols represent the walls of the maze, and the periods (.) represent squares in the possible paths through the maze.

There's a simple algorithm for walking through a maze that guarantees finding the exit (assuming there's an exit). If there's not an exit, you'll 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'll arrive at the exit of the maze. There may be a shorter path than the one you have taken, but you're 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.

7.23 *(Generating Mazes Randomly)* Write a function `mazeGenerator` that takes as an argument a two-dimensional 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 7.22 using several randomly generated mazes.

7.24 *(Mazes of Any Size)* Generalize functions `mazeTraverse` and `mazeGenerator` of Exercises 7.22–7.23 to process mazes of any width and height.

7.25 *(Arrays of Pointers to Functions)* Rewrite the program of Fig. 6.22 to use a menu-driven interface. The program should offer the user four 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 that 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 index into the array for calling each function.

7.26 What does this program do, assuming that the user enters two strings of the same length?

```
// ex07_26.c
// What does this program do?
#include <stdio.h>
#define SIZE 80

int mystery3(const char *s1, const char *s2); // prototype

int main(void)
{
   char string1[SIZE]; // create char array
   char string2[SIZE]; // create char array

   puts("Enter two strings: ");
   scanf("%79s%79s", string1 , string2);
   printf("The result is %d\n", mystery3(string1, string2));
}

int mystery3(const char *s1, const char *s2)
{
   int result = 1;

   for (; *s1 != '\0' && *s2 != '\0'; ++s1, ++s2) {
      if (*s1 != *s2) {
         result = 0;
      }
   }

   return result;
}
```

Special Section: Building Your Own Computer

In the next several exercises, 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!

7.27 ***(Machine-Language Programming)*** Let's create a computer we'll call the Simpletron. As its name implies, it's a simple machine, but as we'll soon see, it's 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` and so on. 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, so instructions are signed four-digit decimal numbers. We 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 *opera-*

tion code, which specifies the operation to be performed. SML operation codes are summarized in Fig. 7.32.

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

Fig. 7.32 | 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's consider several simple SML programs. The following 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 1 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)*

The following 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.

Example 2 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)*

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

a) Use a sentinel-controlled loop to read positive integers 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.

7.28 *(A Computer Simulator)* It may at first seem outrageous, but in this problem you're going to build your own computer. No, you won't be soldering components together. Rather, you'll use the powerful technique of *software-based simulation* to create a *software model* of the Simpletron. You'll not be disappointed. Your Simpletron simulator will turn the computer you're using into a Simpletron, and you'll actually be able to run, test and debug the SML programs you wrote in Exercise 7.27.

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 one-dimensional array `memory` that has 100 elements. Now assume that the simulator is running, and let's examine the dialog as we enter the program of Example 2 of Exercise 7.27:

```
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 the SML program. It begins with the instruction in location `00` and 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, if an instrucion has an `operand`, it's 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's "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*, and so on). A `switch` differentiates among the twelve operations of SML.

The `switch` statement simulates the behavior of various SML instructions 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 ***
```

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*. To help you program your dump function, a sample dump format is shown in Fig. 7.33. A dump after executing a Simpletron program would show the actual values of instructions and data values at the moment execution terminated. You can print leading 0s in front of an integer that is shorter than its field width by placing the 0 formatting flag before the field width in the format specifier as in `"%02d"`. You can place a + or - sign before a value with the + formatting flag. So to produce a number of the form +0000, you can use the format specifier `"%+05d"`.

```
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. 7.33 | Sample Simpletron dump format.

Let's proceed with the execution of our program's first instruction, namely the `+1009` in location `00`. As we've 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. Because 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.

Now let's 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 the SML programs you wrote in Exercise 7.27. 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 a correct number is entered.

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

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

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

Implementation Note: When you implement the Simpletron Simulator, define the `memory` array and all the registers as variables in `main`. The program should contain three other functions—`load`, `execute` and `dump`. Function `load` reads the SML instructions from the user at the keyboard. (Once you study file processing in Chapter 11, you'll be able to read the SML instruction from a file.) Function `execute` executes the SML program currently loaded in the `memory` array. Function `dump` displays the contents of `memory` and all of the registers stored in `main`'s variables. Pass the `memory` array and registers to the other functions as necessary to complete their tasks. Functions `load` and `execute` need to modify variables that are defined in `main`, so you'll need to pass those variables to the functions by reference using pointers. So, you'll need to modify the statements we showed throughout this problem description to use the appropriate pointer notations.

7.29 ***(Modifications to the Simpletron Simulator)*** In Exercise 7.28, 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 12.25 and 12.26, 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 remainder 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 it 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 prints a string beginning at a specified Simpletron memory location. The first half of the word at that location is the length of the string in characters. 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.]

Array of Function Pointer Exercises

7.30 ***(Calculating Circle Circumference, Circle Area or Sphere Volume Using Function Pointers)*** Using the techniques you learned in Fig. 7.28, create a text-based, menu-driven program that allows the user to choose whether to calculate the circumference of a circle, the area of a circle or the volume of a sphere. The program should then input a radius from the user, perform the appropriate calculation and display the result. Use an array of function pointers in which each pointer represents a function that returns `void` and receives a `double` parameter. The corresponding functions should each display messages indicating which calculation was performed, the value of the radius and the result of the calculation.

7.31 ***(Calculator Using Function Pointers)*** Using the techniques you learned in Fig. 7.28, create a text-based, menu-driven program that allows the user to choose whether to add, subtract, multiply or divide two numbers. The program should then input two double values from the user, perform the appropriate calculation and display the result. Use an array of function pointers in which each pointer represents a function that returns `void` and receives two `double` parameters. The corresponding functions should each display messages indicating which calculation was performed, the values of the parameters and the result of the calculation.

Making a Difference

7.32 ***(Polling)*** The Internet and the web are enabling more people to network, join a cause, voice opinions, and so on. The U.S. presidential candidates in 2008 used the Internet intensively to get out their messages and raise money for their campaigns. In this exercise, you'll write a simple polling program that allows users to rate five social-consciousness issues from 1 (least important) to 10 (most important). Pick five causes that are important to you (e.g., political issues, global environmental issues). Use a one-dimensional array `topics` (of type `char *`) to store the five causes. To summarize the survey responses, use a 5-row, 10-column two-dimensional array `responses` (of type

`int`), each row corresponding to an element in the `topics` array. When the program runs, it should ask the user to rate each issue. Have your friends and family respond to the survey. Then have the program display a summary of the results, including:

a) A tabular report with the five topics down the left side and the 10 ratings across the top, listing in each column the number of ratings received for each topic.
b) To the right of each row, show the average of the ratings for that issue.
c) Which issue received the highest point total? Display both the issue and the point total.
d) Which issue received the lowest point total? Display both the issue and the point total.

7.33 ***(Carbon Footprint Calculator: Arrays of Function Pointers)*** Using arrays of function pointers, as you learned in this chapter, you can specify a set of functions that are called with the same types of arguments and return the same type of data. Governments and companies worldwide are becoming increasingly concerned with carbon footprints (annual releases of carbon dioxide into the atmosphere) from buildings burning various types of fuels for heat, vehicles burning fuels for power, and the like. Many scientists blame these greenhouse gases for the phenomenon called global warming. Create three functions that help calculate the carbon footprint of a building, a car and a bicycle, respectively. Each function should input appropriate data from the user, then calculate and display the carbon footprint. (Check out a few websites that explain how to calculate carbon footprints.) Each function should receive no parameters and return `void`. Write a program that prompts the user to enter the type of carbon footprint to calculate, then calls the corresponding function in the array of function pointers. For each type of carbon footprint, display some identifying information and the object's carbon footprint.

Parte II

Pointers (Kochan)

11

Pointers

In this chapter, you examine one of the most sophisticated features of the C programming language: *pointers*. In fact, the power and flexibility that C provides in dealing with pointers serve to set it apart from many other programming languages. Pointers enable you to effectively represent complex data structures, to change values passed as arguments to functions, to work with memory that has been allocated "dynamically" (see Chapter 17, "Miscellaneous and Advanced Features"), and to more concisely and efficiently deal with arrays.

To understand the way in which pointers operate, it is first necessary to understand the concept of *indirection*. You are familiar with this concept from your everyday life. For example, suppose you need to buy a new ink cartridge for your printer. In the company that you work for, all purchases are handled by the Purchasing department. So, you call Jim in Purchasing and ask him to order the new cartridge for you. Jim, in turn, calls the local supply store to order the cartridge. This approach to obtain your new cartridge is actually an indirect one because you are not ordering the cartridge directly from the supply store yourself.

This same notion of indirection applies to the way pointers work in C. A pointer provides an indirect means of accessing the value of a particular data item. And just as there are reasons why it makes sense to go through the Purchasing department to order new cartridges (you don't have to know which particular store the cartridges are being ordered from, for example), so are there good reasons why, at times, it makes sense to use pointers in C.

Defining a Pointer Variable

But enough talk—it's time to see how pointers actually work. Suppose you define a variable called `count` as follows:

```
int  count = 10;
```

You can define another variable, called `int_pointer`, that can be used to enable you to indirectly access the value of `count` by the declaration

```
int  *int_pointer;
```

The asterisk defines to the C system that the variable `int_pointer` is of type *pointer to* `int`. This means that `int_pointer` is used in the program to indirectly access the value of one or more integer values.

You have seen how the `&` operator was used in the `scanf` calls of previous programs. This unary operator, known as the *address* operator, is used to make a pointer to an object in C. So, if `x` is a variable of a particular type, the expression `&x` is a pointer to that variable. The expression `&x` can be assigned to any pointer variable, if desired, that has been declared to be a pointer to the same type as `x`.

Therefore, with the definitions of `count` and `int_pointer` as given, you can write a statement such as

```
int_pointer = &count;
```

to set up the indirect reference between `int_pointer` and `count`. The address operator has the effect of assigning to the variable `int_pointer`, not the value of `count`, but a *pointer* to the variable `count`. The link that has been made between `int_pointer` and `count` is conceptualized in Figure 11.1. The directed line illustrates the idea that `int_pointer` does not directly contain the value of `count`, but a pointer to the variable `count`.

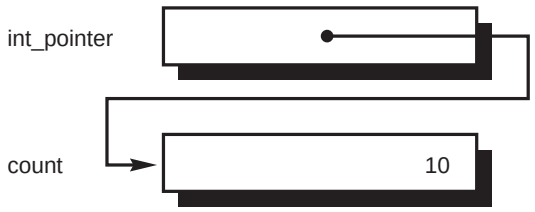


Figure 11.1 Pointer to an integer.

To reference the contents of `count` through the pointer variable `int_pointer`, you use the *indirection* operator, which is the asterisk `*`. So, if `x` is defined as type `int`, the statement

```
x = *int_pointer;
```

assigns the value that is indirectly referenced through `int_pointer` to the variable `x`. Because `int_pointer` was previously set pointing to `count`, this statement has the effect of assigning the value contained in the variable `count`—which is `10`—to the variable `x`.

The previous statements have been incorporated into Program 11.1, which illustrates the two fundamental pointer operators: the address operator, `&`, and the indirection operator, `*`.

Program 11.1 **Illustrating Pointers**

```
// Program to illustrate pointers

#include <stdio.h>

int main (void)
{
     int   count = 10, x;
     int   *int_pointer;

     int_pointer = &count;
     x = *int_pointer;

     printf ("count = %i, x = %i\n", count, x);

     return 0;
}
```

Program 11.1 **Output**

```
count = 10, x = 10
```

The variables `count` and `x` are declared to be integer variables in the normal fashion. On the next line, the variable `int_pointer` is declared to be of type "pointer to `int`." Note that the two lines of declarations could have been combined into the single line

```
int  count = 10, x, *int_pointer;
```

Next, the address operator is applied to the variable `count`. This has the effect of creating a pointer to this variable, which is then assigned by the program to the variable `int_pointer`.

Execution of the next statement in the program,

```
x = *int_pointer;
```

proceeds as follows: The indirection operator tells the C system to treat the variable `int_pointer` as containing a pointer to another data item. This pointer is then used to access the desired data item, whose type is specified by the declaration of the pointer variable. Because you told the compiler that `int_pointer` points to integers when you declared the variable, the compiler knows that the value referenced by the expression `*int_pointer` is an integer. And because you set `int_pointer` to point to the integer variable `count` in the previous program statement, it is the value of `count` that is indirectly accessed by this expression.

You should realize that Program 11.1 is a manufactured example of the use of pointers and does not show a practical use for them in a program. Such motivation is presented shortly, after you have become familiar with the basic ways in which pointers can be defined and manipulated in a program.

Program 11.2 illustrates some interesting properties of pointer variables. Here, a pointer to a character is used.

Program 11.2 **More Pointer Basics**

```
// Further examples of pointers

#include <stdio.h>

int main (void)
{
     char  c = 'Q';
     char  *char_pointer = &c;

     printf ("%c %c\n", c, *char_pointer);

     c = '/';
     printf ("%c %c\n", c, *char_pointer);

     *char_pointer = '(';
     printf ("%c %c\n", c, *char_pointer);

     return 0;
}
```

Program 11.2 **Output**

```
Q Q
/ /
( (
```

The character variable `c` is defined and initialized to the character `'Q'`. In the next line of the program, the variable `char_pointer` is defined to be of type "pointer to `char`," meaning that whatever value is stored inside this variable should be treated as an indirect reference (pointer) to a character. Notice that you can assign an initial value to this variable in the normal fashion. The value that you assign to `char_pointer` in the program is a pointer to the variable `c`, which is obtained by applying the address operator to the variable `c`. (Note that this initialization generates a compiler error if `c` had been defined *after* this statement because a variable must always be declared *before* its value can be referenced in an expression.)

The declaration of the variable `char_pointer` and the assignment of its initial value could have been equivalently expressed in two separate statements as

```
char  *char_pointer;
char_pointer = &c;
```

(and *not* by the statements

```
char  *char_pointer;
*char_pointer = &c;
```

as might be implied from the single-line declaration).

Always remember, that the value of a pointer in C is meaningless until it is set pointing to something.

The first `printf` call simply displays the contents of the variable `c` and the contents of the variable that is referenced by `char_pointer`. Because you set `char_pointer` to point to the variable `c`, the value that is displayed is the contents of `c`, as verified by the first line of the program's output.

In the next line of the program, the character `'/'` is assigned to the character variable `c`. Because `char_pointer` still points to the variable `c`, displaying the value of `*char_pointer` in the subsequent `printf` call correctly displays this new value of `c` at the terminal. This is an important concept. Unless the value of `char_pointer` is changed, the expression `*char_pointer` *always* accesses the value of `c`. So, as the value of `c` changes, so does the value of `*char_pointer`.

The previous discussion can help you to understand how the program statement that appears next in the program works. Unless `char_pointer` is changed, the expression `*char_pointer` always references the value of `c`. Therefore, in the expression

```
*char_pointer = '(';
```

you are assigning the left parenthesis character to `c`. More formally, the character `'('` is assigned to the variable that is pointed to by `char_pointer`. You know that this variable is `c` because you placed a pointer to `c` in `char_pointer` at the beginning of the program.

The preceding concepts are the key to your understanding of the operation of pointers. Please review them at this point if they still seem a bit unclear.

Using Pointers in Expressions

In Program 11.3, two integer pointers, `p1` and `p2`, are defined. Notice how the value referenced by a pointer can be used in an arithmetic expression. If `p1` is defined to be of type "pointer to integer," what conclusion do you think can be made about the use of `*p1` in an expression?

Program 11.3 **Using Pointers in Expressions**

```
// More on pointers

#include <stdio.h>

int main (void)
{
```

Program 11.3 **Continued**

```
    int  i1, i2;
    int  *p1, *p2;

    i1 = 5;
    p1 = &i1;
    i2 = *p1 / 2 + 10;
    p2 = p1;

    printf ("i1 = %i, i2 = %i, *p1 = %i, *p2 = %i\n", i1, i2, *p1, *p2);

    return 0;
}
```

Program 11.3 **Output**

```
i1 = 5, i2 = 12, *p1 = 5, *p2 = 5
```

After defining the integer variables `i1` and `i2` and the integer pointer variables `p1` and `p2`, the program then assigns the value `5` to `i1` and stores a pointer to `i1` inside `p1`. Next, the value of `i2` is calculated with the following expression:

```
i2 = *p1 / 2 + 10;
```

In As implied from the discussions of Program 11.2, if a pointer `px` points to a variable `x`, and `px` has been defined to be a pointer to the same data type as is `x`, then use of `*px` in an expression is, in all respects, identical to the use of `x` in the same expression.

Because in Program 11.3 the variable `p1` is defined to be an integer pointer, the preceding expression is evaluated using the rules of integer arithmetic. And because the value of `*p1` is `5` (`p1` points to `i1`), the final result of the evaluation of the preceding expression is `12`, which is the value that is assigned to `i2`. (The pointer reference operator `*` has higher precedence than the arithmetic operation of division. In fact, this operator, as well as the address operator `&`, has higher precedence than *all* binary operators in C.)

In the next statement, the value of the pointer `p1` is assigned to `p2`. This assignment is perfectly valid and has the effect of setting `p2` to point to the same data item to which `p1` points. Because `p1` points to `i1`, after the assignment statement has been executed, `p2` *also* points to `i1` (and you can have as many pointers to the same item as you want in C).

The `printf` call verifies that the values of `i1`, `*p1`, and `*p2` are all the same (`5`) and that the value of `i2` was set to `12` by the program.

Working with Pointers and Structures

You have seen how a pointer can be defined to point to a basic data type, such as an `int` or a `char`. But pointers can also be defined to point to structures. In Chapter 9, "Working with Structures," you defined your `date` structure as follows:

```
struct date
{
     int  month;
     int  day;
     int  year;
};
```

Just as you defined variables to be of type `struct date`, as in

```
struct date   todaysDate;
```

so can you define a variable to be a pointer to a `struct date` variable:

```
struct date  *datePtr;
```

The variable `datePtr`, as just defined, then can be used in the expected fashion. For example, you can set it to point to `todaysDate` with the assignment statement

```
datePtr = &todaysDate;
```

After such an assignment has been made, you then can indirectly access any of the members of the `date` structure pointed to by `datePtr` in the following way:

```
(*datePtr).day = 21;
```

This statement has the effect of setting the day of the `date` structure pointed to by `datePtr` to 21. The parentheses are required because the structure member operator `.` has higher precedence than the indirection operator `*`.

To test the value of `month` stored in the `date` structure pointed to by `datePtr`, a statement such as

```
if  ( (*datePtr).month == 12  )
          ...
```

can be used.

Pointers to structures are so often used in C that a special operator exists in the language. The structure pointer operator `->`, which is the dash followed by the greater than sign, permits expressions that would otherwise be written as

```
(*x).y
```

to be more clearly expressed as

```
x->y
```

So, the previous `if` statement can be conveniently written as

```
if  ( datePtr->month == 12 )
     ...
```

Program 9.1, the first program that illustrated structures, was rewritten using the concept of structure pointers, as shown in Program 11.4.

Program 11.4 **Using Pointers to Structures**

```
//  Program to illustrate structure pointers

#include <stdio.h>

int main (void)
{
     struct date
     {
          int  month;
          int  day;
          int  year;
     };

     struct date  today, *datePtr;

     datePtr = &today;

     datePtr->month = 9;
     datePtr->day = 25;
     datePtr->year = 2004;

     printf ("Today's date is %i/%i/%.2i.\n",
               datePtr->month, datePtr->day, datePtr->year % 100);

     return 0;
}
```

Program 11.4 **Output**

```
Today's date is 9/25/04.
```

Figure 11.2 depicts how the variables `today` and `datePtr` would look after all of the assignment statements from the preceding program have been executed.

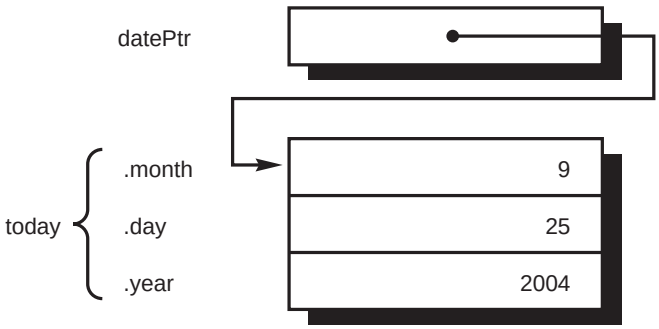


Figure 11.2 Pointer to a structure.

Once again, it should be pointed out that there is no real motivation shown here as to why you should even bother using a structure pointer when it seems as though you can get along just fine without it (as you did in Program 9.1). You will discover the motivation shortly.

Structures Containing Pointers

Naturally, a pointer also can be a member of a structure. In the structure definition

```
struct  intPtrs
{
     int  *p1;
     int  *p2;
};
```

a structure called `intPtrs` is defined to contain two integer pointers, the first one called `p1` and the second one `p2`. You can define a variable of type `struct intPtrs` in the usual way:

```
struct intPtrs  pointers;
```

The variable `pointers` can now be used in the normal fashion, remembering that `pointers` itself is *not* a pointer, but a structure variable that has two pointers as its members.

Program 11.5 shows how the `intPtrs` structure can be handled in a C program.

Program 11.5 **Using Structures Containing Pointers**

```
// Function to use structures containing pointers

#include <stdio.h>

int main (void)
{
     struct  intPtrs
     {
          int  *p1;
          int  *p2;
     };

     struct intPtrs  pointers;
     int  i1 = 100, i2;

     pointers.p1 = &i1;
     pointers.p2 = &i2;
     *pointers.p2 = -97;
```

Program 11.5 **Continued**

```
    printf ("i1 = %i, *pointers.p1 = %i\n", i1, *pointers.p1);
    printf ("i2 = %i, *pointers.p2 = %i\n", i2, *pointers.p2);
    return 0;
}
```

Program 11.5 **Output**

```
i1 = 100, *pointers.p1 = 100
i2 = -97, *pointers.p2 = -97
```

After the variables have been defined, the assignment statement

```
pointers.p1 = &i1;
```

sets the `p1` member of `pointers` pointing to the integer variable `i1`, whereas the next statement

```
pointers.p2 = &i2;
```

sets the `p2` member pointing to `i2`. Next, `-97` is assigned to the variable that is pointed to by `pointers.p2`. Because you just set this to point to `i2`, `-97` is stored in `i2`. No parentheses are needed in this assignment statement because, as mentioned previously, the structure member operator `.` has higher precedence than the indirection operator. Therefore, the pointer is correctly referenced from the structure *before* the indirection operator is applied. Of course, parentheses could have been used just to play it safe, as at times it can be difficult to try to remember which of two operators has higher precedence.

The two `printf` calls that follow each other in the preceding program verify that the correct assignments were made.

Figure 11.3 has been provided to help you understand the relationship between the variables `i1`, `i2`, and `pointers` after the assignment statements from Program 11.5 have been executed. As you can see in Figure 11.3, the `p1` member points to the variable `i1`, which contains the value `100`, whereas the `p2` member points to the variable `i2`, which contains the value `-97`.

Linked Lists

The concepts of pointers to structures and structures containing pointers are very powerful ones in C, for they enable you to create sophisticated data structures, such as *linked lists*, *doubly linked lists*, and *trees*.

Suppose you define a structure as follows:

```
struct entry
{
    int            value;
    struct entry   *next;
};
```

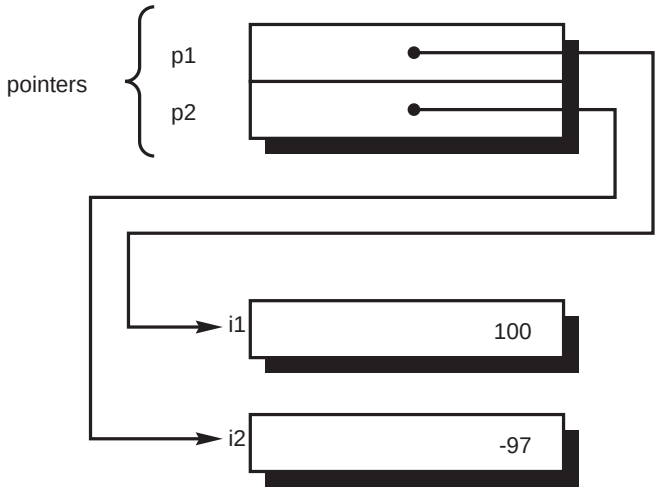


Figure 11.3 Structure containing pointers.

This defines a structure called `entry`, which contains two members. The first member of the structure is a simple integer called `value`. The second member of the structure is a member called `next`, which is a *pointer to an* `entry` *structure*. Think about this for a moment. Contained inside an `entry` structure is a pointer to another `entry` structure. This is a perfectly valid concept in the C language. Now suppose you define two variables to be of type `struct entry` as follows:

```
struct entry  n1, n2;
```

You set the next pointer of structure `n1` to point to structure `n2` by executing the following statement:

```
n1.next = &n2;
```

This statement effectively makes a link between `n1` and `n2`, as depicted in Figure 11.4.

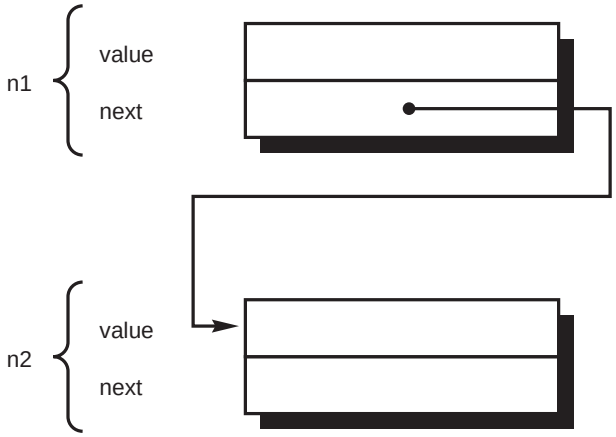


Figure 11.4 Linked structures.

Assuming a variable `n3` were also defined to be of type `struct entry`, you could add another link with the following statement:

```
n2.next = &n3;
```

This resulting chain of linked entries, known more formally as a *linked list*, is illustrated in Figure 11.5. Program 11.6 illustrates this linked list.

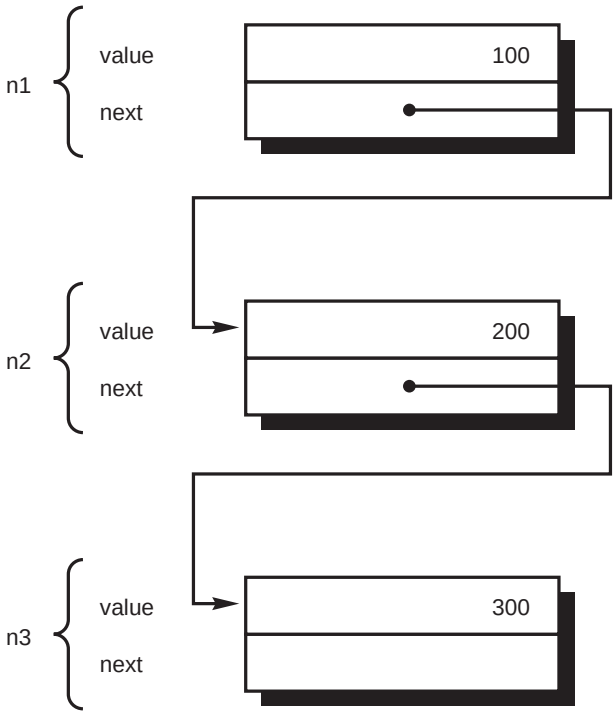


Figure 11.5 A linked list.

Program 11.6 **Using Linked Lists**

```
// Function to use linked Lists

#include <stdio.h>

int main (void)
{
     struct  entry
     {
          int           value;
          struct entry  *next;
     };

     struct entry n1, n2, n3;
     int          i;
```

Program 11.6 **Continued**

```
    n1.value = 100;
    n2.value = 200;
    n3.value = 300;

    n1.next = &n2;
    n2.next = &n3;

    i = n1.next->value;
    printf ("%i  ", i);

    printf ("%i\n", n2.next->value);

    return 0;
}
```

Program 11.6 **Output**

```
200  300
```

The structures `n1`, `n2`, and `n3` are defined to be of type `struct entry`, which consists of an integer member called `value` and a pointer to an `entry` structure called `next`. The program then assigns the values `100`, `200`, and `300` to the value members of `n1`, `n2`, and `n3`, respectively.

The next two statements in the program

```
n1.next = &n2;
n2.next = &n3;
```

set up the linked list, with the `next` member of `n1` pointing to `n2` and the `next` member of `n2` pointing to `n3`.

Execution of the statement

```
i = n1.next->value;
```

proceeds as follows: The `value` member of the `entry` structure pointed to by `n1.next` is accessed and assigned to the integer variable `i`. Because you set `n1.next` to point to `n2`, the `value` member of `n2` is accessed by this statement. Therefore, this statement has the net result of assigning 200 to `i`, as verified by the `printf` call that follows in the program. You might want to verify that the expression `n1.next->value` is the correct one to use and not `n1.next.value`, because the `n1.next` field contains a pointer to a structure, and not the structure itself. This distinction is important and can quickly lead to programming errors if it is not fully understood.

The structure member operator `.` and the structure pointer operator `->` have the same precedence in the C language. In expressions such as the preceding one, where

both operators are used, the operators are evaluated from left to right. Therefore, the expression is evaluated as

```
i = (n1.next)->value;
```

which is what was intended.

The second `printf` call in Program 11.6 displays the `value` member that is pointed to by `n2.next`. Because you set `n2.next` to point to `n3`, the contents of `n3.value` are displayed by the program.

As mentioned, the concept of a linked list is a very powerful one in programming. Linked lists greatly simplify operations such as the insertion and removal of elements from large lists of sorted items.

For example, if `n1`, `n2`, and `n3` are as defined previously, you can easily remove `n2` from the list simply by setting the `next` field of `n1` to point to whatever `n2` is pointing to:

```
n1.next = n2.next;
```

This statement has the effect of copying the pointer contained in `n2.next` into `n1.next`, and, because `n2.next` was set to point to `n3`, `n1.next` is now pointing to `n3`. Furthermore, because `n1` no longer points to `n2`, you have effectively removed it from your list. Figure 11.6 depicts this situation after the preceding statement is executed. Of course, you could have set `n1` pointing to `n3` directly with the statement

```
n1.next = &n3;
```

but this latter statement is not as general because you must know in advance that `n2` is pointing to `n3`.

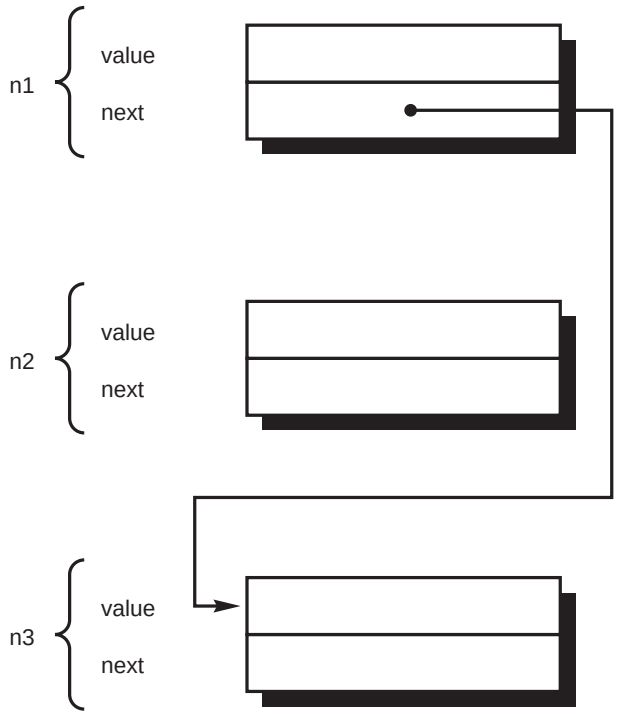


Figure 11.6 Removing an entry from a linked list.

Inserting an element into a list is just as straightforward. If you want to insert a `struct entry` called `n2_3` after `n2` in the list, you can simply set `n2_3.next` to point to whatever `n2.next` was pointing to, and then set `n2.next` to point to `n2_3`. So, the sequence of statements

```
n2_3.next = n2.next;
n2.next = &n2_3;
```

inserts `n2_3` into the list, immediately after entry `n2`. Note that the sequence of the preceding statements is important because executing the second statement first overwrites the pointer stored in `n2.next` before it has a chance to be assigned to `n2_3.next`. The inserted element `n2_3` is depicted in Figure 11.7. Notice that `n2_3` is not shown between `n1` and `n3`. This is to emphasize that `n2_3` can be anywhere in memory and does not have to physically occur after `n1` and before `n3`. This is one of the main motivations for the use of a linked list approach for storing information: Entries of the list do not have to be stored sequentially in memory, as is the case with elements in an array.

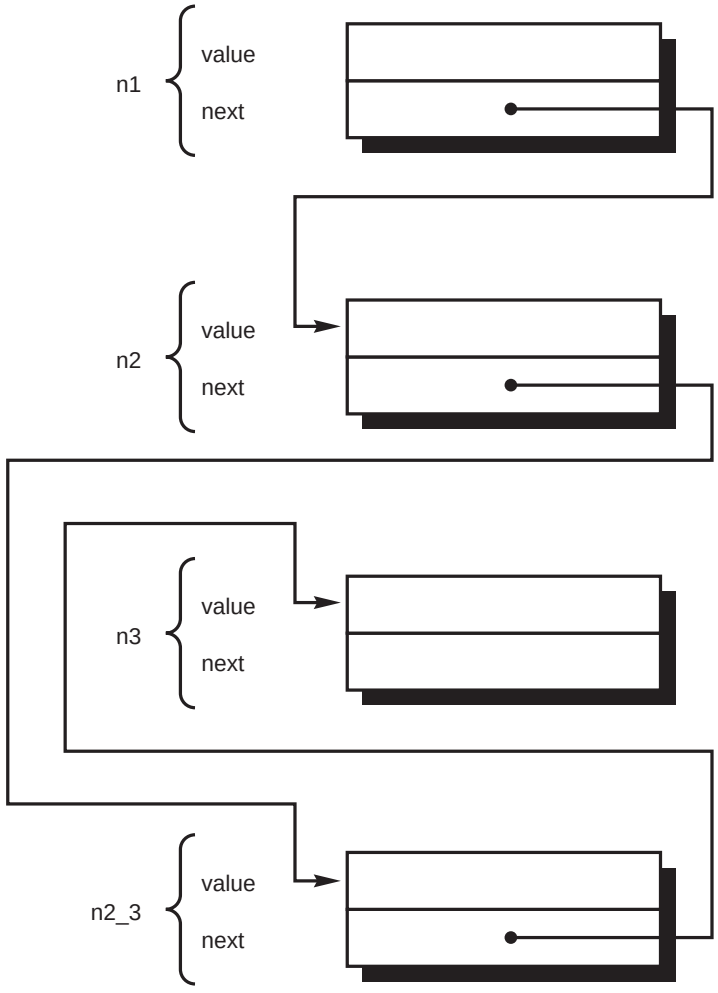


Figure 11.7 Inserting an element into a linked list.

Before developing some functions to work with linked lists, two more issues must be discussed. Usually associated with a linked list is at least one pointer to the list. Often, a pointer to the start of the list is kept. So, for your original three-element list, which consisted of `n1`, `n2`, and `n3`, you can define a variable called `list_pointer` and set it to point to the beginning of the list with the statement

```
struct entry *list_pointer = &n1;
```

assuming that `n1` has been previously defined. A pointer to a list is useful for sequencing through the entries in the list, as you see shortly.

The second issue to be discussed involves the idea of having some way to identify the end of the list. This is needed so that a procedure that searches through the list, for example, can tell when it has reached the final element in the list. By convention, a constant value of `0` is used for such a purpose and is known as the *null* pointer. You can use the null pointer to mark the end of a list by storing this value in the pointer field of the last entry of the list.[1]

In your three-entry list, you can mark its end by storing the null pointer in `n3.next`:

```
n3.next =  (struct entry *) 0;
```

You see in Chapter 13, "The Preprocessor," how this assignment statement can be made a bit more readable.

The type cast operator is used to cast the constant 0 to the appropriate type ("pointer to `struct entry`"). It's not required, but makes the statement more readable.

Figure 11.8 depicts the linked list from Program 11.6, with a `struct entry` pointer called `list_pointer` pointing to the start of the list and the `n3.next` field set to the null pointer.

Program 11.7 incorporates the concepts just described. The program uses a `while` loop to sequence through the list and display the `value` member of each entry in the list.

Program 11.7 **Traversing a Linked List**

```
// Program to traverse a linked list

#include <stdio.h>

int main (void)
{
     struct entry
     {
          int            value;
          struct entry   *next;
     };
```

1. A null pointer is not necessarily internally represented as the value `0`. However, the compiler must recognize assignment of the constant 0 to a pointer as assigning the null pointer. This also applies to comparing a pointer against the constant 0: The compiler interprets it as a test to see if the pointer is null.

Program 11.7 **Continued**

```
    struct entry   n1, n2, n3;
    struct entry   *list_pointer = &n1;

    n1.value = 100;
    n1.next  = &n2;

    n2.value = 200;
    n2.next  = &n3;

    n3.value = 300;
    n3.next  = (struct entry *) 0;    // Mark list end with null pointer

    while ( list_pointer != (struct entry *) 0 ) {
         printf ("%i\n", list_pointer->value);
         list_pointer = list_pointer->next;
    }

    return 0;
}
```

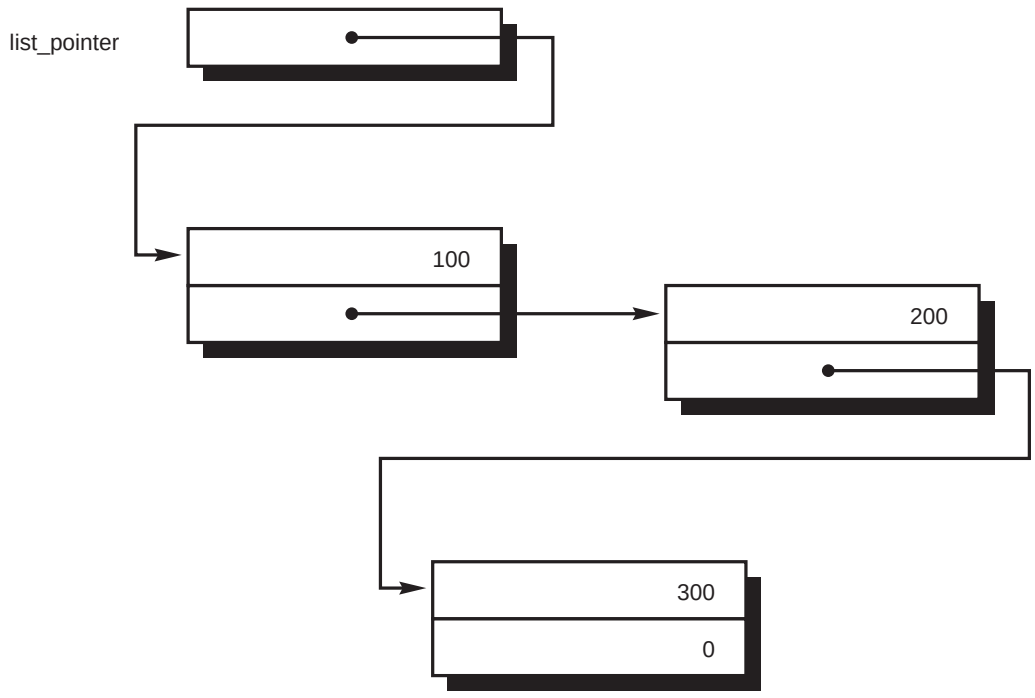


Figure 11.8 Linked list showing list pointer and terminating null.

Program 11.7 **Output**

```
100
200
300
```

The program defines the variables `n1`, `n2`, and `n3` and the pointer variable `list_pointer`, which is initially set to point to `n1`, the first entry in the list. The next program statements link together the three elements of the list, with the next member of `n3` set to the null pointer to mark the end of the list.

A `while` loop is then set up to sequence through each element of the list. This loop is executed as long as the value of `list_pointer` is not equal to the null pointer. The `printf` call inside the `while` loop displays the `value` member of the entry currently pointed to by `list_pointer`.

The statement that follows the `printf` call,

```
list_pointer = list_pointer->next;
```

has the effect of taking the pointer from the `next` member of the structure pointed to by `list_pointer` and assigning it to `list_pointer`. So, the first time through the loop, this statement takes the pointer contained in `n1.next` (remember, `list_pointer` was initially set pointing to `n1`) and assigns it to `list_pointer`. Because this value is not null—it's a pointer to the `entry` structure `n2`—the `while` loop is repeated.

The second time through, the `while` loop results in the display of `n2.value`, which is `200`. The `next` member of `n2` is then copied into `list_pointer`, and because you set this value to point to `n3`, `list_pointer` points to `n3` by the end of the second pass through the loop.

When the `while` loop is executed the third time, the `printf` call displays the value of `300` as contained in `n3.value`. At that point, `list_pointer->next` (which is actually `n3.next`) is copied into `list_pointer`, and, because you set this member to the null pointer, the `while` loop terminates after it has been executed three times.

Trace through the operation of the `while` loop just discussed, using a pencil and paper, if necessary, to keep track of the values of the various variables. Understanding the operation of this loop is the key to your understanding the operation of pointers in C. Incidentally, it should be noted that this same `while` loop can be used to sequence through the elements of a list of *any* size, provided the end of the list is marked by the null pointer.

When working with actual linked lists in a program, you will not normally link together list entries that have been explicitly defined like in the program examples in this section. You did that here just to illustrate the mechanics of working with a linked list. In actual practice, you will typically ask the system to give you memory for each new list entry and you will link it into the list while the program is executing. This is done with a mechanism known as *dynamic memory allocation*, and is covered in Chapter 17.

The Keyword const and Pointers

You have seen how a variable or an array can be declared as `const` to alert the compiler as well as the reader that the contents of a variable or an array will not be changed by the program. With pointers, there are two things to consider: whether the pointer will be changed, and whether the value that the pointer points to will be changed. Think about that for a second. Assume the following declarations:

```
char c = 'X';
char *charPtr = &c;
```

The pointer variable `charPtr` is set pointing to the variable `c`. If the pointer variable is always set pointing to `c`, it can be declared as a `const` pointer as follows:

```
char * const charPtr = &c;
```

(Read this as "`charPtr` is a constant pointer to a character.") So, a statement like this:

```
charPtr = &d;    // not valid
```

causes the GNU C compiler to give a message like this:[2]

```
foo.c:10: warning: assignment of read-only variable 'charPtr'
```

Now if, instead, the location pointed to by `charPtr` will not change *through the pointer variable* `charPtr`, that can be noted with a declaration as follows:

```
const char *charPtr = &c;
```

(Read this as "`charPtr` points to a constant character.") Now of course, that doesn't mean that the value cannot be changed by the variable `c`, which is what `charPtr` is set pointing to. It means, however, that it won't be changed with a subsequent statement like this:

```
*charPtr = 'Y';    // not valid
```

which causes the GNU C compiler to issue a message like this:

```
foo.c:11: warning: assignment of read-only location
```

In the case in which both the pointer variable and the location it points to will not be changed through the pointer, the following declaration can be used:

```
const char * const *charPtr = &c;
```

The first use of `const` says the contents of the location the pointer references will not be changed. The second use says that the pointer itself will not be changed. Admittedly, this looks a little confusing, but it's worth noting at this point in the text.[3]

2. Your compiler may give a different warning message, or no message at all.

3. The keyword `const` is not used in every program example where it can be employed; only in selected examples. Until you are familiar with reading expressions such as previously shown, it can make understanding the examples more difficult.

Pointers and Functions

Pointers and functions get along quite well together. That is, you can pass a pointer as an argument to a function in the normal fashion, and you can also have a function return a pointer as its result.

The first case cited previously, passing pointer arguments, is straightforward enough: The pointer is simply included in the list of arguments to the function in the normal fashion. So, to pass the pointer `list_pointer` from the previous program to a function called `print_list`, you can write

```
print_list (list_pointer);
```

Inside the `print_list` routine, the formal parameter must be declared to be a pointer to the appropriate type:

```
void  print_list  (struct entry  *pointer)
{
     ...
}
```

The formal parameter `pointer` can then be used in the same way as a normal pointer variable. One thing worth remembering when dealing with pointers that are sent to functions as arguments: The value of the pointer is copied into the formal parameter when the function is called. Therefore, any change made to the formal parameter by the function does *not* affect the pointer that was passed to the function. But here's the catch: Although the pointer cannot be changed by the function, the data elements that the pointer references *can* be changed! Program 11.8 helps clarify this point.

Program 11.8 **Using Pointers and Functions**

```
// Program to illustrate using pointers and functions

#include <stdio.h>

void test (int  *int_pointer)
{
     *int_pointer = 100;
}

int main (void)
{
     void test (int  *int_pointer);
     int  i = 50, *p = &i;

     printf ("Before the call to test i = %i\n", i);
```

Program 11.8 **Continued**

```
     test (p);
     printf ("After the call to test i = %i\n", i);

     return 0;
}
```

Program 11.8 **Output**

```
Before the call to test i = 50
After the call to test i = 100
```

The function `test` is defined to take as its argument a pointer to an integer. Inside the function, a single statement is executed to set the integer pointed to by `int_pointer` to the value `100`.

The `main` routine defines an integer variable `i` with an initial value of `50` and a pointer to an integer called `p` that is set to point to `i`. The program then displays the value of `i` and calls the `test` function, passing the pointer `p` as the argument. As you can see from the second line of the program's output, the `test` function did, in fact, change the value of `i` to `100`.

Now consider Program 11.9.

Program 11.9 **Using Pointers to Exchange Values**

```
// More on pointers and functions

#include <stdio.h>

void  exchange (int * const pint1, int * const pint2)
{
     int  temp;

     temp = *pint1;
     *pint1 = *pint2;
     *pint2 = temp;
}

int main (void)
{
     void  exchange (int * const pint1, int * const pint2);
     int   i1 = -5, i2 = 66, *p1 = &i1, *p2 = &i2;

     printf ("i1 = %i, i2 = %i\n", i1, i2);
```

Program 11.9 **Continued**

```
    exchange (p1, p2);
    printf ("i1 = %i, i2 = %i\n", i1, i2);

    exchange (&i1, &i2);
    printf ("i1 = %i, i2 = %i\n", i1, i2);

    return 0;
}
```

Program 11.9 **Output**

```
i1 = -5, i2 = 66
i1 = 66, i2 = -5
i1 = -5, i2 = 66
```

The purpose of the `exchange` function is to interchange the two integer values pointed to by its two arguments. The function header

```
void  exchange (int * const pint1, int * const pint2)
```

says that the `exchange` function takes two integer pointers as arguments, and that the pointers will not be changed by the function (the use of the keyword `const`).

The local integer variable `temp` is used to hold one of the integer values while the exchange is made. Its value is set equal to the integer that is pointed to by `pint1`. The integer pointed to by `pint2` is then copied into the integer pointed to by `pint1`, and the value of `temp` is then stored in the integer pointed to by `pint2`, thus making the exchange complete.

The `main` routine defines integers `i1` and `i2` with values of `-5` and `66`, respectively. Two integer pointers, `p1` and `p2`, are then defined and are set to point to `i1` and `i2`, respectively. The program then displays the values of `i1` and `i2` and calls the `exchange` function, passing the two pointers, `p1` and `p2`, as arguments. The `exchange` function exchanges the value contained in the integer pointed to by `p1` with the value contained in the integer pointed to by `p2`. Because `p1` points to `i1`, and `p2` to `i2`, the values of `i1` and `i2` end up getting exchanged by the function. The output from the second `printf` call verifies that the exchange worked properly.

The second call to `exchange` is a bit more interesting. This time, the arguments that are passed to the function are pointers to `i1` and `i2` that are manufactured right on the spot by applying the address operator to these two variables. Because the expression `&i1` produces a pointer to the integer variable `i1`, this is right in line with the type of argument that your function expects for the first argument (a pointer to an integer). The same applies for the second argument as well. And as can be seen from the program's output, the `exchange` function did its job and switched the values of `i1` and `i2` back to their original values.

You should realize that without the use of pointers, you could not have written your `exchange` function to exchange the value of two integers because you are limited to returning only a single value from a function and because a function cannot permanently change the value of its arguments. Study Program 11.9 in detail. It illustrates with a small example the key concepts to be understood when dealing with pointers in C.

Program 11.10 shows how a function can return a pointer. The program defines a function called `findEntry` whose purpose is to search through a linked list to find a specified value. When the specified value is found, the program returns a pointer to the entry in the list. If the desired value is not found, the program returns the null pointer.

Program 11.10 **Returning a Pointer from a Function**

```
#include <stdio.h>

struct entry
{
     int  value;
     struct entry   *next;
};

struct entry  *findEntry (struct entry  *listPtr, int match)
{
     while  ( listPtr != (struct entry *) 0 )
          if ( listPtr->value == match )
               return (listPtr);
          else
               listPtr = listPtr->next;

     return (struct entry *) 0;
}

int main (void)
{
    struct entry  *findEntry (struct entry  *listPtr, int match);
    struct entry  n1, n2, n3;
    struct entry  *listPtr, *listStart = &n1;

    int search;

    n1.value = 100;
    n1.next =  &n2;

    n2.value = 200;
    n2.next =  &n3;

    n3.value = 300;
    n3.next =  0;
```

Program 11.10 **Continued**

```
    printf ("Enter value to locate: ");
    scanf ("%i", &search);

    listPtr = findEntry (listStart, search);

    if ( listPtr != (struct entry *) 0 )
        printf ("Found %i.\n", listPtr->value);
    else
        printf ("Not found.\n");

    return 0;
}
```

Program 11.10 **Output**

```
Enter value to locate: 200
Found 200.
```

Program 11.10 **Output (Rerun)**

```
Enter value to locate: 400
Not found.
```

Program 11.10 **Output (Second Rerun)**

```
Enter value to locate: 300
Found 300.
```

The function header

```
struct entry  *findEntry (struct entry  *listPtr, int match)
```

specifies that the function `findEntry` returns a pointer to an `entry` structure and that it takes such a pointer as its first argument and an integer as its second. The function begins by entering a `while` loop to sequence through the elements of the list. This loop is executed until either `match` is found equal to one of the `value` entries in the list (in which case the value of `listPtr` is immediately returned) or until the null pointer is reached (in which case the `while` loop is exited and a null pointer is returned).

After setting up the list as in previous programs, the `main` routine asks the user for a value to locate in the list and then calls the `findEntry` function with a pointer to the start of the list (`listStart`) and the value entered by the user (`search`) as arguments. The pointer that is returned by `findEntry` is assigned to the `struct entry` pointer variable `listPtr`. If `listPtr` is not null, the `value` member pointed to by `listPtr` is

displayed. This should be the same as the value entered by the user. If `listPtr` is null, then a "Not found." message is displayed.

The program's output verifies that the values `200` and `300` were correctly located in the list, and the value `400` was not found because it did not, in fact, exist in the list.

The pointer that is returned by the `findEntry` function in the program does not seem to serve any useful purpose. However, in more practical situations, this pointer might be used to access other elements contained in the particular entry of the list. For example, you could have a linked list of your dictionary entries from Chapter 10, "Character Strings." Then, you could call the `findEntry` function (or rename it `lookup` as it was called in that chapter) to search the linked list of dictionary entries for the given word. The pointer returned by the `lookup` function could then be used to access the `definition` member of the entry.

Organizing the dictionary as a linked list has several advantages. Inserting a new word into the dictionary is easy: After determining where in the list the new entry is to be inserted, it can be done by simply adjusting some pointers, as illustrated earlier in this chapter. Removing an entry from the dictionary is also simple. Finally, as you learn in Chapter 17, this approach also provides the framework that enables you to dynamically expand the size of the dictionary.

However, the linked list approach for the organization of the dictionary does suffer from one major drawback: You cannot apply your fast binary search algorithm to such a list. This algorithm only works with an array of elements that can be directly indexed. Unfortunately, there is no faster way to search your linked list other than by a straight, sequential search because each entry in the list can only be accessed from the previous one.

One way to glean the benefits of easy insertion and removal of elements, as well as fast search time, is by using a different type of data structure known as a *tree*. Other approaches, such as using *hash tables*, are also feasible. The reader is respectfully referred elsewhere—such as to *The Art of Computer Programming, Volume 3, Sorting and Searching* (Donald E. Knuth, Addison-Wesley)—for discussion of these types of data structures, which can be easily implemented in C with the techniques already described.

Pointers and Arrays

One of the most common uses of pointers in C is as pointers to arrays. The main reasons for using pointers to arrays are ones of notational convenience and of program efficiency. Pointers to arrays generally result in code that uses less memory and executes faster. The reason for this will become apparent through our discussions in this section.

If you have an array of 100 integers called `values`, you can define a pointer called `valuesPtr`, which can be used to access the integers contained in this array with the statement

```
int  *valuesPtr;
```

When you define a pointer that is used to point to the elements of an array, you don't designate the pointer as type "pointer to array"; rather, you designate the pointer as pointing to the type of element that is contained in the array.

If you have an array of characters called `text`, you could similarly define a pointer to be used to point to elements in `text` with the statement

```
char  *textPtr;
```

To set `valuesPtr` to point to the first element in the `values` array, you simply write

```
valuesPtr = values;
```

The address operator is not used in this case because the C compiler treats the appearance of an array name without a subscript as a pointer to the array. Therefore, simply specifying `values` without a subscript has the effect of producing a pointer to the first element of `values` (see Figure 11.9).

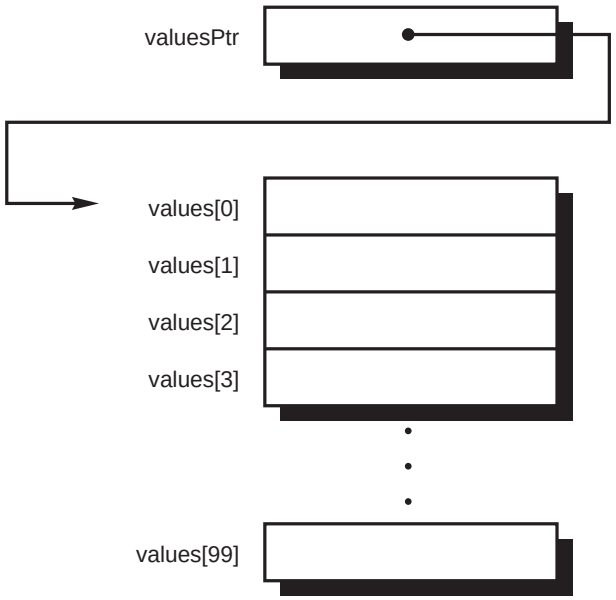


Figure 11.9 Pointer to an array element.

An equivalent way of producing a pointer to the start of `values` is to apply the address operator to the first element of the array. Thus, the statement

```
valuesPtr = &values[0];
```

can be used to serve the same purpose as placing a pointer to the first element of `values` in the pointer variable `valuesPtr`.

To set `textPtr` to point to the first character inside the `text` array, either the statement

```
textPtr = text;
```

or

```
textPtr = &text[0];
```

can be used. Whichever statement you choose to use is strictly a matter of taste.

The real power of using pointers to arrays comes into play when you want to sequence through the elements of an array. If `valuesPtr` is as previously defined and is set pointing to the first element of `values`, the expression

```
*valuesPtr
```

can be used to access the first integer of the `values` array, that is, `values[0]`. To reference `values[3]` through the `valuesPtr` variable, you can add 3 to `valuesPtr` and then apply the indirection operator:

```
*(valuesPtr + 3)
```

In general, the expression

```
*(valuesPtr + i)
```

can be used to access the value contained in `values[i]`.

So, to set `values[10]` to 27, you could obviously write the expression

```
values[10] = 27;
```

or, using `valuesPtr`, you could write

```
*(valuesPtr + 10) = 27;
```

To set `valuesPtr` to point to the second element of the `values` array, you can apply the address operator to `values[1]` and assign the result to `valuesPtr`:

```
valuesPtr = &values[1];
```

If `valuesPtr` points to `values[0]`, you can set it to point to `values[1]` by simply adding 1 to the value of `valuesPtr`:

```
valuesPtr += 1;
```

This is a perfectly valid expression in C and can be used for pointers to *any* data type.

So, in general, if `a` is an array of elements of type `x`, `px` is of type "pointer to `x`," and `i` and `n` are integer constants or variables, the statement

```
px = a;
```

sets `px` to point to the first element of `a`, and the expression

```
*(px + i)
```

subsequently references the value contained in `a[i]`. Furthermore, the statement

```
px += n;
```

sets px to point n elements farther in the array, *no matter what type of element is contained in the array*.

The increment and decrement operators ++ and -- are particularly handy when dealing with pointers. Applying the increment operator to a pointer has the same effect as adding one to the pointer, while applying the decrement operator has the same effect as subtracting one from the pointer. So, if textPtr is defined as a char pointer and is set pointing to the beginning of an array of chars called text, the statement

```
++textPtr;
```

sets textPtr pointing to the next character in text, which is text[1]. In a similar fashion, the statement

```
--textPtr;
```

sets textPtr pointing to the previous character in text, assuming, of course, that textPtr was not pointing to the beginning of text prior to the execution of this statement.

It is perfectly valid to compare two pointer variables in C. This is particularly useful when comparing two pointers in the same array. For example, you can test the pointer valuesPtr to see if it points past the end of an array containing 100 elements by comparing it to a pointer to the last element in the array. So, the expression

```
valuesPtr > &values[99]
```

is TRUE (nonzero) if valuesPtr is pointing past the last element in the values array, and is FALSE (zero) otherwise. Recall from previous discussions that you can replace the preceding expression with its equivalent

```
valuesPtr > values + 99
```

because values used without a subscript is a pointer to the beginning of the values array. (Remember, it's the same as writing &values[0].)

Program 11.11 illustrates pointers to arrays. The arraySum function calculates the sum of the elements contained in an array of integers.

Program 11.11 **Working with Pointers to Arrays**

```
// Function to sum the elements of an integer array

#include <stdio.h>

int  arraySum (int  array[], const int  n)
{
     int  sum = 0, *ptr;
     int  * const arrayEnd = array + n;

     for ( ptr = array;  ptr < arrayEnd;  ++ptr )
          sum += *ptr;
```

Program 11.11 **Continued**

```
    return sum;
}

int main (void)
{
    int  arraySum (int  array[], const int  n);
    int  values[10] = { 3, 7, -9, 3, 6, -1, 7, 9, 1, -5 };

    printf ("The sum is %i\n", arraySum (values, 10));

    return 0;
}
```

Program 11.11 **Output**

```
The sum is 21
```

Inside the `arraySum` function, the constant integer pointer `arrayEnd` is defined and set pointing immediately after the last element of `array`. A `for` loop is then set up to sequence through the elements of `array`. The value of `ptr` is set to point to the beginning of `array` when the loop is entered. Each time through the loop, the element of `array` pointed to by `ptr` is added into `sum`. The value of `ptr` is then incremented by the `for` loop to set it pointing to the next element in `array`. When `ptr` points past the end of `array`, the `for` loop is exited, and the value of `sum` is returned to the calling routine.

A Slight Digression About Program Optimization

It is pointed out that the local variable `arrayEnd` was not actually needed by the function because you could have explicitly compared the value of `ptr` to the end of the array inside the `for` loop:

```
for ( ...; pointer <= array + n; ... )
```

The sole motivation for using `arrayEnd` was one of optimization. Each time through the `for` loop, the looping conditions are evaluated. Because the expression `array + n` is never changed from within the loop, its value is constant throughout the execution of the `for` loop. By evaluating it once *before* the loop is entered, you save the time that would otherwise be spent reevaluating this expression each time through the loop. Although there is virtually no savings in time for a 10-element array, especially if the `arraySum` function is called only once by the program, there could be a more substantial savings if this function were heavily used by a program for summing large-sized arrays, for example.

The other issue to be discussed about program optimization concerns the very use of pointers themselves in a program. In the `arraySum` function discussed earlier, the

expression `*ptr` is used inside the `for` loop to access the elements in the array. Formerly, you would have written your `arraySum` function with a `for` loop that used an index variable, such as `i`, and then would have added the value of `array[i]` into `sum` inside the loop. In general, the process of indexing an array takes more time to execute than does the process of accessing the contents of a pointer. In fact, this is one of the main reasons why pointers are used to access the elements of an array—the code that is generated is generally more efficient. Of course, if access to the array is not generally sequential, pointers accomplish nothing, as far as this issue is concerned, because the expression `*(pointer + j)` takes just as long to execute as does the expression `array[j]`.

Is It an Array or Is It a Pointer?

Recall that to pass an array to a function, you simply specify the name of the array, as you did previously with the call to the `arraySum` function. You should also remember that to produce a pointer to an array, you need only specify the name of the array. This implies that in the call to the `arraySum` function, what was passed to the function was actually a *pointer* to the array `values`. This is precisely the case and explains why you are able to change the elements of an array from within a function.

But if it is indeed the case that a pointer to the array is passed to the function, then you might wonder why the formal parameter inside the function isn't declared to be a pointer. In other words, in the declaration of `array` in the `arraySum` function, why isn't the declaration

```
int  *array;
```

used? Shouldn't all references to an array from within a function be made using pointer variables?

To answer these questions, recall the previous discussion about pointers and arrays. As mentioned, if `valuesPtr` points to the same type of element as contained in an array called `values`, the expression `*(valuesPtr + i)` is in all ways equivalent to the expression `values[i]`, assuming that `valuesPtr` has been set to point to the beginning of `values`. What follows from this is that you also can use the expression `*(values + i)` to reference the `i`th element of the array `values`, and, in general, if `x` is an array of any type, the expression `x[i]` can always be equivalently expressed in C as `*(x + i)`.

As you can see, pointers and arrays are intimately related in C, and this is why you can declare `array` to be of type "array of `int`s" inside the `arraySum` function *or* to be of type "pointer to `int`." Either declaration works just fine in the preceding program—try it and see.

If you are going to be using index numbers to reference the elements of an array that is passed to a function, declare the corresponding formal parameter to be an array. This more correctly reflects the use of the array by the function. Similarly, if you are using the argument as a pointer to the array, declare it to be of type pointer.

Realizing now that you could have declared `array` to be an `int` pointer in the preceding program example, and then could have subsequently used it as such, you can

eliminate the variable `ptr` from the function and use `array` instead, as shown in Program 11.12.

Program 11.12 **Summing the Elements of an Array**

```
// Function to sum the elements of an integer array  Ver. 2

#include <stdio.h>

int  arraySum (int  *array, const int  n)
{
     int  sum = 0;
     int  * const arrayEnd = array + n;

     for (  ; array < arrayEnd;  ++array )
          sum += *array;

     return sum;
}



int main (void)
{
     int  arraySum (int  *array, const int  n);
     int  values[10] = { 3, 7, -9, 3, 6, -1, 7, 9, 1, -5 };

     printf ("The sum is %i\n", arraySum (values, 10));

     return 0;
}
```

Program 11.12 **Output**

```
The sum is 21
```

The program is fairly self-explanatory. The first expression inside the `for` loop was omitted because no value had to be initialized before the loop was started. One point worth repeating is that when the `arraySum` function is called, a pointer to the `values` array is passed, where it is called `array` inside the function. Changes to the value of `array` (as opposed to the values referenced by `array`) do not in any way affect the contents of the `values` array. So, the increment operator that is applied to `array` is just incrementing a pointer to the array `values`, and not affecting its contents. (Of course, you know that you *can* change values in the array if you want to, simply by assigning values to the elements referenced by the pointer.)

Pointers to Character Strings

One of the most common applications of using a pointer to an array is as a pointer to a character string. The reasons are ones of notational convenience and efficiency. To show how easily pointers to character strings can be used, write a function called `copyString` to copy one string into another. If you write this function using normal array indexing methods, the function might be coded as follows:

```
void copyString (char  to[], char  from[])
{
     int  i;

     for ( i = 0;  from[i] != '\0';  ++i )
          to[i] = from[i];

     to[i] = '\0';
}
```

The `for` loop is exited before the null character is copied into the `to` array, thus explaining the need for the last statement in the function.

If you write `copyString` using pointers, you no longer need the index variable `i`. A pointer version is shown in Program 11.13.

Program 11.13 **Pointer Version of** `copyString`

```
#include <stdio.h>

void copyString (char  *to, char  *from)
{
     for (  ;  *from != '\0';  ++from, ++to )
          *to = *from;

     *to = '\0';
}

int main (void)
{
     void  copyString (char  *to, char  *from);
     char  string1[] = "A string to be copied.";
     char  string2[50];

     copyString (string2, string1);
     printf ("%s\n", string2);

     copyString (string2, "So is this.");
     printf ("%s\n", string2);

     return 0;
}
```

Program 11.13 **Output**

```
A string to be copied.
So is this.
```

The `copyString` function defines the two formal parameters `to` and `from` as character pointers and not as character arrays as was done in the previous version of `copyString`. This reflects how these two variables are used by the function.

A `for` loop is then entered (with no initial conditions) to copy the string pointed to by `from` into the string pointed to by `to`. Each time through the loop, the `from` and `to` pointers are each incremented by one. This sets the `from` pointer pointing to the next character that is to be copied from the source string and sets the `to` pointer pointing to the location in the destination string where the next character is to be stored.

When the `from` pointer points to the null character, the `for` loop is exited. The function then places the null character at the end of the destination string.

In the `main` routine, the `copyString` function is called twice, the first time to copy the contents of `string1` into `string2`, and the second time to copy the contents of the constant character string `"So is this."` into `string2`.

Constant Character Strings and Pointers

The fact that the call

```
copyString (string2, "So is this.");
```

works in the previous program implies that when a constant character string is passed as an argument to a function, what is actually passed is a pointer to that character string. Not only is this true in this case, but it also can be generalized by saying that *whenever* a constant character string is used in C, it is a pointer to that character string that is produced. So, if `textPtr` is declared to be a character pointer, as in

```
char  *textPtr;
```

then the statement

```
textPtr = "A character string.";
```

assigns to `textPtr` a *pointer* to the constant character string `"A character string."` Be careful to make the distinction here between character pointers and character arrays, as the type of assignment just shown is *not* valid with a character array. So, for example, if `text` is defined instead to be an array of `char`s, with a statement such as

```
char  text[80];
```

then you *could not* write a statement such as

```
text = "This is not valid.";
```

The *only* time that C lets you get away with performing this type of assignment to a character array is when initializing it, as in

```
char  text[80] =  "This is okay.";
```

Initializing the `text` array in this manner does not have the effect of storing a pointer to the character string `"This is okay."` inside `text`, but rather the actual characters themselves inside corresponding elements of the `text` array.

If `text` is a character pointer, initializing `text` with the statement

```
char  *text =  "This is okay.";
```

assigns to it a pointer to the character string `"This is okay."`

As another example of the distinction between character strings and character string pointers, the following sets up an array called `days`, which contains *pointers* to the names of the days of the week.

```
char *days[] =
    { "Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
      "Saturday" };
```

The array `days` is defined to contain seven entries, each a pointer to a character string. So `days[0]` contains a pointer to the character string `"Sunday"`, `days[1]` contains a pointer to the string `"Monday"`, and so on (see Figure 11.10). You could display the name of the third weekday, for example, with the following statement:

```
printf ("%s\n", days[3]);
```

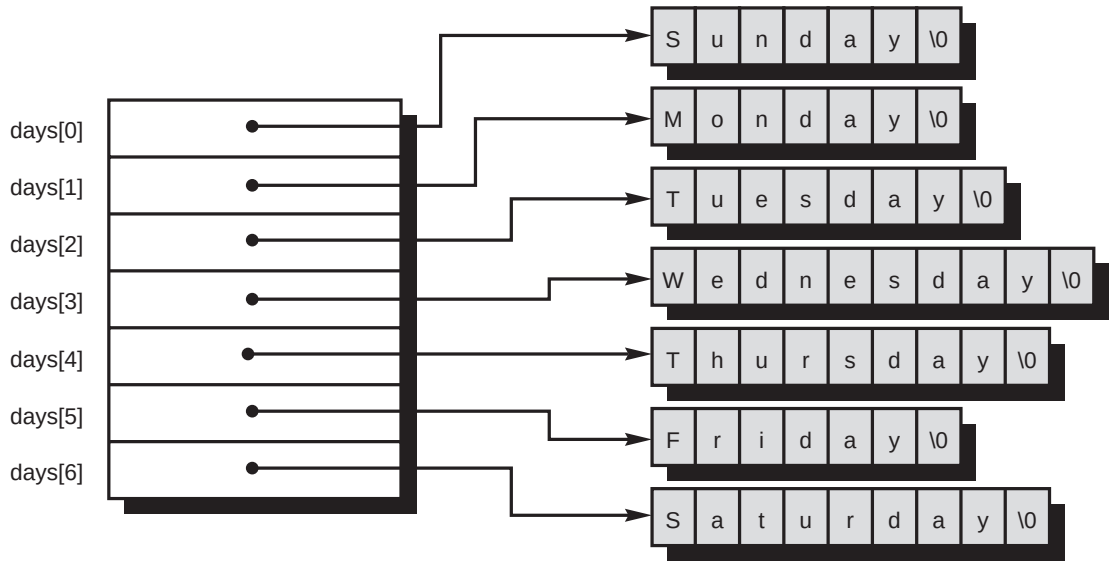


Figure 11.10 Array of pointers.

The Increment and Decrement Operators Revisited

Up to this point, whenever you used the increment or decrement operator, it was the only operator that appeared in the expression. When you write the expression `++x`, you know that this has the effect of adding 1 to the value of the variable `x`. And as you have

just seen, if x is a pointer to an array, this has the effect of setting x to point to the next element of the array.

The increment and decrement operators can be used in expressions in which other operators also appear. In such cases, it becomes important to know more precisely how these operators work.

So far, when you used the increment and decrement operators, you always placed them *before* the variables that were being incremented or decremented. So, to increment a variable i, you simply wrote

```
++i;
```

Actually, it also is perfectly valid to place the increment operator *after* the variable, as follows:

```
i++;
```

Both expressions are perfectly valid and both achieve the same result—namely, of incrementing the value of i. In the first case, where the ++ is placed before its operand, the increment operation is more precisely identified as a *preincrement*. In the second case, where the ++ is placed after its operand, the operation is identified as a *postincrement*.

The same discussion applies to the decrement operator. So the statement

```
--i;
```

technically performs a *predecrement* of i, whereas the statement

```
i--;
```

performs a *postdecrement* of i. Both have the same net result of subtracting 1 from the value of i.

It is when the increment and decrement operators are used in more complex expressions that the distinction between the *pre-* and *post-* nature of these operators is realized.

Suppose you have two integers called i and j. If you set the value of i to 0 and then write the statement

```
j = ++i;
```

the value that gets assigned to j is 1, and not 0 as you might expect. In the case of the preincrement operator, the variable is incremented *before* its value is used in the expression. So, in the preceding expression, the value of i is first incremented from 0 to 1 and then its value is assigned to j, as if the following two statements had been written instead:

```
++i;
j = i;
```

If you instead use the postincrement operator in the statement

```
j = i++;
```

then `i` is incremented *after* its value has been assigned to `j`. So, if `i` is `0` before the preceding statement is executed, `0` is assigned to `j` and *then* `i` is incremented by 1, as if the statements

```
j = i;
++i;
```

were used instead. As another example, if `i` is equal to 1, then the statement

```
x = a[--i];
```

has the effect of assigning the value of `a[0]` to `x` because the variable `i` is decremented before its value is used to index into `a`. The statement

```
x = a[i--];
```

used instead has the effect of assigning the value of `a[1]` to `x` because `i` is decremented after its value has been used to index into `a`.

As a third example of the distinction between the pre- and post- increment and decrement operators, the function call

```
printf ("%i\n", ++i);
```

increments `i` and then sends its value to the `printf` function, whereas the call

```
printf ("%i\n", i++);
```

increments `i` after its value has been sent to the function. So, if `i` is equal to 100, the first `printf` call displays 101, whereas the second `printf` call displays 100. In either case, the value of `i` is equal to 101 after the statement has executed.

As a final example on this topic before presenting Program 11.14, if `textPtr` is a character pointer, the expression

```
*(++textPtr)
```

first increments `textPtr` and then fetches the character it points to, whereas the expression

```
*(textPtr++)
```

fetches the character pointed to by `textPtr` before its value is incremented. In either case, the parentheses are not required because the `*` and `++` operators have equal precedence but associate from right to left.

Now go back to the `copyString` function from Program 11.13 and rewrite it to incorporate the increment operations directly into the assignment statement.

Because the `to` and `from` pointers are incremented each time after the assignment statement inside the `for` loop is executed, they should be incorporated into the assignment statement as postincrement operations. The revised `for` loop of Program 11.13 then becomes

```
for (  ;  *from != '\0';  )
     *to++ = *from++;
```

Execution of the assignment statement inside the loop proceeds as follows. The character pointed to by `from` is retrieved and then `from` is incremented to point to the next character in the source string. The referenced character is then stored inside the location pointed to by `to`, and then `to` is incremented to point to the next location in the destination string.

Study the preceding assignment statement until you fully understand its operation. Statements of this type are so commonly used in C programs, it's important that you understand it completely before continuing.

The preceding `for` statement hardly seems worthwhile because it has no initial expression and no looping expression. In fact, the logic would be better served when expressed in the form of a `while` loop. This has been done in Program 11.14. This program presents your new version of the `copyString` function. The `while` loop uses the fact that the null character is equal to the value `0`, as is commonly done by experienced C programmers.

Program 11.14 **Revised Version of the `copyString` Function**

```
// Function to copy one string to another. Pointer Ver. 2

#include <stdio.h>

void  copyString (char  *to, char  *from)
{
     while ( *from )
          *to++ = *from++;

     *to = '\0';
}



int main (void)
{
     void  copyString (char  *to, char  *from);
     char  string1[] = "A string to be copied.";
     char  string2[50];

     copyString (string2, string1);
     printf ("%s\n", string2);

     copyString (string2, "So is this.");
     printf ("%s\n", string2);

     return 0;
}
```

Program 11.14 **Output**

```
A string to be copied.
So is this.
```

Operations on Pointers

As you have seen in this chapter, you can add or subtract integer values from pointers. Furthermore, you can compare two pointers to see if they are equal or not, or if one pointer is less than or greater than another pointer. The only other operation that is permitted on pointers is the subtraction of two pointers of the same type. The result of subtracting two pointers in C is the number of elements contained between the two pointers. So, if `a` points to an array of elements of any type and `b` points to another element somewhere farther along in the same array, the expression `b - a` represents the number of elements between these two pointers. For example, if `p` points to some element in an array `x`, the statement

```
n = p - x;
```

has the effect of assigning to the variable `n` (assumed here to be an integer variable) the index number of the element inside `x` to which `p` points.[4] Therefore, if `p` is set pointing to the hundredth element in `x` by a statement such as

```
p = &x[99];
```

the value of `n` after the previous subtraction is performed is `99`.

As a practical application of this newly learned fact about pointer subtraction, take a look at a new version of the `stringLength` function from Chapter 10.

In Program 11.15, the character pointer `cptr` is used to sequence through the characters pointed to by `string` until the null character is reached. At that point, `string` is subtracted from `cptr` to obtain the number of elements (characters) contained in the string. The program's output verifies that the function is working correctly.

Program 11.15 **Using Pointers to Find the Length of a String**

```
// Function to count the characters in a string - Pointer version

#include <stdio.h>

int  stringLength (const char  *string)
{
     const char  *cptr = string;
```

4. The actual type of signed integer that is produced by subtracting two pointers (for example, `int`, `long int`, or `long long int`) is `ptrdiff_t`, which is defined in the standard header file `<stddef.h>`.

Program 11.15 **Continued**

```
     while ( *cptr )
          ++cptr;
     return  cptr - string;
}

int main (void)
{
     int  stringLength (const char  *string);

     printf ("%i  ", stringLength ("stringLength test"));
     printf ("%i  ", stringLength (""));
     printf ("%i\n", stringLength ("complete"));

     return 0;
}
```

Program 11.15 **Output**

```
17  0  8
```

Pointers to Functions

Of a slightly more advanced nature, but presented here for the sake of completeness, is the notion of a pointer to a function. When working with pointers to functions, the C compiler needs to know not only that the pointer variable points to a function, but also the type of value returned by that function as well as the number and types of its arguments. To declare a variable `fnPtr` to be of type "pointer to function that returns an `int` and that takes no arguments," the declaration

```
int  (*fnPtr) (void);
```

can be written. The parentheses around `*fnPtr` are required because otherwise the C compiler treats the preceding statement as the declaration of a function called `fnPtr` that returns a pointer to an `int` (because the function call operator `()` has higher precedence than the pointer indirection operator `*`).

To set your function pointer pointing to a specific function, you simply assign the name of the function to it. So, if `lookup` is a function that returns an `int` and that takes no arguments, the statement

```
fnPtr = lookup;
```

stores a pointer to this function inside the function pointer variable `fnPtr`. Writing a function name without a subsequent set of parentheses is treated in an analogous way to writing an array name without a subscript. The C compiler automatically produces a

pointer to the specified function. An ampersand is permitted in front of the function name, but it's not required.

If the `lookup` function has not been previously defined in the program, it is necessary to declare the function before the preceding assignment can be made. So, a statement such as

```
int  lookup (void);
```

is needed before a pointer to this function can be assigned to the variable `fnPtr`.

You can call the function that is indirectly referenced through a pointer variable by applying the function call operator to the pointer, listing any arguments to the function inside the parentheses. For example,

```
entry = fnPtr ();
```

calls the function pointed to by `fnPtr`, storing the returned value inside the variable `entry`.

One common application for pointers to functions is in passing them as arguments to other functions. The standard C library uses this, for example, in the function `qsort`, which performs a "quicksort" on an array of data elements. This function takes as one of its arguments a pointer to a function that is called whenever `qsort` needs to compare two elements in the array being sorted. In this manner, `qsort` can be used to sort arrays of any type, as the actual comparison of any two elements in the array is made by a user-supplied function, and not by the `qsort` function itself. Appendix B, "The Standard C Library," goes into more detail about `qsort` and contains an actual example of its use.

Another common application for function pointers is to create what is known as *dispatch* tables. You can't store functions themselves inside the elements of an array. However, it is valid to store function *pointers* inside an array. Given this, you can create tables that contain pointers to functions to be called. For example, you might create a table for processing different commands that will be entered by a user. Each entry in the table could contain both the command name and a pointer to a function to call to process that particular command. Now, whenever the user enters a command, you can look up the command inside the table and invoke the corresponding function to handle it.

Pointers and Memory Addresses

Before ending this discussion of pointers in C, you should note the details of how they are actually implemented. A computer's memory can be conceptualized as a sequential collection of storage cells. Each cell of the computer's memory has a number, called an *address*, associated with it. Typically, the first address of a computer's memory is numbered 0. On most computer systems, a "cell" is called a *byte*.

The computer uses memory for storing the instructions of your computer program, and also for storing the values of the variables that are associated with a program. So, if you declare a variable called `count` to be of type `int`, the system assigns location(s) in

memory to hold the value of `count` while the program is executing. This location might be at address 500, for example, inside the computer's memory.

Luckily, one of the advantages of higher-level programming languages such as C is that you don't need to concern yourself with the particular memory addresses that are assigned to variables—they are automatically handled by the system. However, the knowledge that a unique memory address is associated with each variable will help you to understand the way pointers operate.

When you apply the address operator to a variable in C, the value that is generated is the actual address of that variable inside the computer's memory. (Obviously, this is where the address operator gets its name.) So, the statement

```
intPtr = &count;
```

assigns to `intPtr` the address in the computer's memory that has been assigned to the variable `count`. So, if `count` is located at address 500 and contains the value `10`, this statement assigns the value `500` to `intPtr`, as depicted in Figure 11.11.

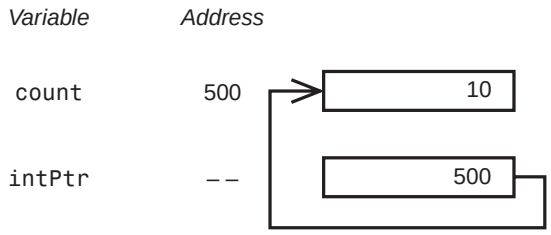


Figure 11.11 Pointers and memory addresses.

The address of `intPtr` is shown in Figure 11.11 as `--` because its actual value is irrelevant for this example.

Applying the indirection operator to a pointer variable, as in the expression

```
*intPtr
```

has the effect of treating the value contained in the pointer variable as a memory address. The value stored at that memory address is then fetched and interpreted in accordance with the type declared for the pointer variable. So, if `intPtr` is of type pointer to `int`, the value stored in the memory address given by `*intPtr` is interpreted as an integer by the system. In our example, the value stored at memory address 500 is fetched and interpreted as an integer. The result of the expression is 10, and is of type `int`.

Storing a value in a location reference by a pointer, as in

```
*intPtr = 20;
```

proceeds in a like manner. The contents of `intPtr` is fetched and treated as a memory address. The specified integer value is then stored at that address. In the preceding statement, the integer value of `20` is, therefore, stored at memory address 500.

At times, system programmers must access particular locations inside the computer's memory. In such cases, this knowledge of the way that pointer variables operate proves helpful.

As you can see from this chapter, the pointer is a very powerful construct in C. The flexibility in defining pointer variables extends beyond those illustrated in this chapter. For example, you can define a pointer to a pointer, and even a pointer to a pointer to a pointer. These types of constructs are beyond the scope of this book, although they are simply logical extensions of everything you've learned about pointers in this chapter.

The topic of pointers is probably the hardest for novices to grasp. You should reread any sections of this chapter that still seem unclear before proceeding. Solving the exercises that follow will also help you to understand the material.

Exercises

1. Type in and run the 15 programs presented in this chapter. Compare the output produced by each program with the output presented after each program in the text.
2. Write a function called `insertEntry` to insert a new entry into a linked list. Have the procedure take as arguments a pointer to the list entry to be inserted (of type `struct entry` as defined in this chapter), and a pointer to an element in the list *after* which the new `entry` is to be inserted.
3. The function developed in exercise 2 only inserts an element after an existing element in the list, thereby preventing you from inserting a new entry at the front of the list. How can you use this same function and yet overcome this problem? (*Hint:* Think about setting up a special structure to point to the beginning of the list.)
4. Write a function called `removeEntry` to remove an `entry` from a linked list. The sole argument to the procedure should be a pointer to the list. Have the function remove the entry *after* the one pointed to by the argument. (Why can't you remove the entry pointed to by the argument?) You need to use the special structure you set up in exercise 3 to handle the special case of removing the first element from the list.
5. A *doubly linked list* is a list in which each entry contains a pointer to the preceding entry in the list as well as a pointer to the next entry in the list. Define the appropriate structure definition for a doubly linked list entry and then write a small program that implements a small doubly linked list and prints out the elements of the list.
6. Develop `insertEntry` and `removeEntry` functions for a doubly linked list that are similar in function to those developed in previous exercises for a singly linked list. Why can your `removeEntry` function now take as its argument a direct pointer to the entry to be removed from the list?
7. Write a pointer version of the `sort` function from Chapter 8, "Working with Functions." Be certain that pointers are exclusively used by the function, including index variables in the loops.

8. Write a function called `sort3` to sort three integers into ascending order. (This function is not to be implemented with arrays.)
9. Rewrite the `readLine` function from Chapter 10 so that it uses a character pointer rather than an array.
10. Rewrite the `compareStrings` function from Chapter 10 to use character pointers instead of arrays.
11. Given the definition of a `date` structure as defined in this chapter, write a function called `dateUpdate` that takes a pointer to a `date` structure as its argument and that updates the structure to the following day (see Program 9.4).
12. Given the following declarations:

```
char  *message = "Programming in C is fun\n";
char  message2[] = "You said it\n";
char  *format  = "x = %i\n";
int   x = 100;
```

determine whether each `printf` call from the following sets is valid and produces the same output as other calls from the set.

```
/*** set 1 ***/
printf ("Programming in C is fun\n");
printf ("%s", "Programming in C is fun\n");
printf ("%s", message);
printf (message);

/*** set 2 ***/
printf ("You said it\n");
printf ("%s", message2);
printf (message2);
printf ("%s", &message2[0]);

/*** set 3 ***/
printf ("said it\n");
printf (message2 + 4);
printf ("%s", message2 + 4);
printf ("%s", &message2[4]);

/*** set 4 ***/
printf ("x = %i\n", x);
printf (format, x);
```