7 C Pointers

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

By indirection find direction out.
—William Shakespeare

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

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

Objectives

In this chapter, you'll learn:

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

Outline

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 simulate pass-by-reference, to pass functions between functions, and to create and manipulate dynamic data structures, i.e., data structures that can grow and shrink at execution time, such as linked lists, queues, stacks and trees. This chapter explains basic pointer concepts. Chapter 10 examines the use of pointers with structures. Chapter 12 introduces dynamic memory management techniques and presents examples of creating and using dynamic data structures.

7.2 Pointer Variable Definitions and Initialization

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

Declaring Pointers

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

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

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.

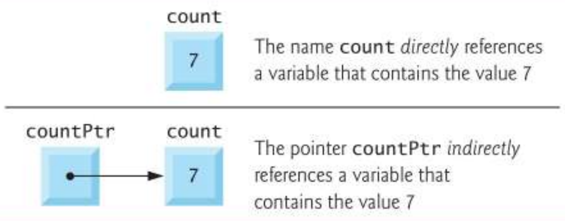

count
7
The name count *directly* references a variable that contains the value 7
countPtr
count
7
The pointer countPtr *indirectly* references a variable that contains the value 7

Fig. 7.1 | Directly and indirectly referencing a variable.

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 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 thus 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. 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

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.

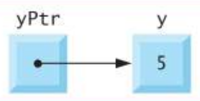


Fig. 7.2 | Graphical representation of a pointer pointing to an integer variable in memory.

Pointer Representation in Memory

Figure 7.3 shows the representation of the 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.

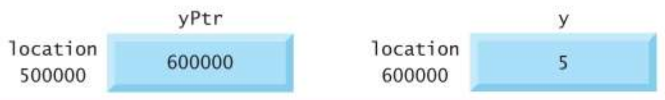


Fig. 7.3 | Representation of `y` and `yPtr` in memory.

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`, namely 5. Using `*` in this manner is called **dereferencing a pointer**.

Common Programming Error 7.2

Dereferencing a pointer that has not been properly initialized or that has not been assigned to point to a specific location in memory 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.) Notice that the *address* of `a` and the *value* of `aPtr` are identical in the output, thus confirming that the address of `a` is indeed assigned to the pointer variable `aPtr` (line 11). The `&` and `*` operators are complements of one another—when they're both applied consecutively to `aPtr` in either order (line 21), the same result is printed. Figure 7.5 lists the precedence and associativity of the operators introduced to this point.

```c
// Fig. 7.4: fig07_04.c
// Using the & and * pointer operators.
#include <stdio.h>

int main( void )
{
   int a; // a is an integer
   int *aPtr; // aPtr is a pointer to an integer

   a = 7;
   aPtr = &a; // set aPtr to the address of a

   printf( "The address of a is %p"
           "\nThe value of aPtr is %p", &a, aPtr );

   printf( "\n\nThe value of a is %d"
           "\nThe value of *aPtr is %d", a, *aPtr );

   printf( "\n\nShowing that * and & are complements of "
            "each other\n&*aPtr = %p"
            "\n*&aPtr = %p\n", &*aPtr, *&aPtr );
} // end main
```

```
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
`<` `<=` `>` `>=`	left to right	relational
`==` `!=`	left to right	equality
`&&`	left to right	logical AND
`||`	left to right	logical OR

Fig. 7.5 | Operator precedence and associativity. (Part 1 of 2.)

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

Fig. 7.5 | Operator precedence and associativity. (Part 2 of 2.)

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. *All arguments in C are passed by value.* As we saw in Chapter 5, `return` may be used to return one value from a called function to a caller (or to return control from a called function without passing back a value). Many functions require the capability to *modify variables in the caller* or to pass a pointer to a large data object to avoid the overhead of passing the object by value (which incurs the time and memory overheads of making a copy of the object).

In C, you use pointers and the indirection operator to *simulate* 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`. Figure 7.6 passes the variable `number` by value to function `cubeByValue` (line 14). The `cubeByValue` function cubes its argument and passes the new value back to `main` using a `return` statement. The new value is assigned to `number` in `main` (line 14).

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

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

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

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

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

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

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

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

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

```
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.)

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.

The function prototype for `cubeByReference` (line 6) contains `int *` in parentheses. 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.

For a function that expects a single-subscripted 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 single-subscripted array. This, of course, means that the function must "know" when it's receiving an array or simply a single variable for which it's to perform pass-by-reference. When the compiler encounters a function parameter for a single-subscripted array of the form `int b[]`, the compiler converts the parameter to the pointer notation `int *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.

7.5 Using the const Qualifier with Pointers

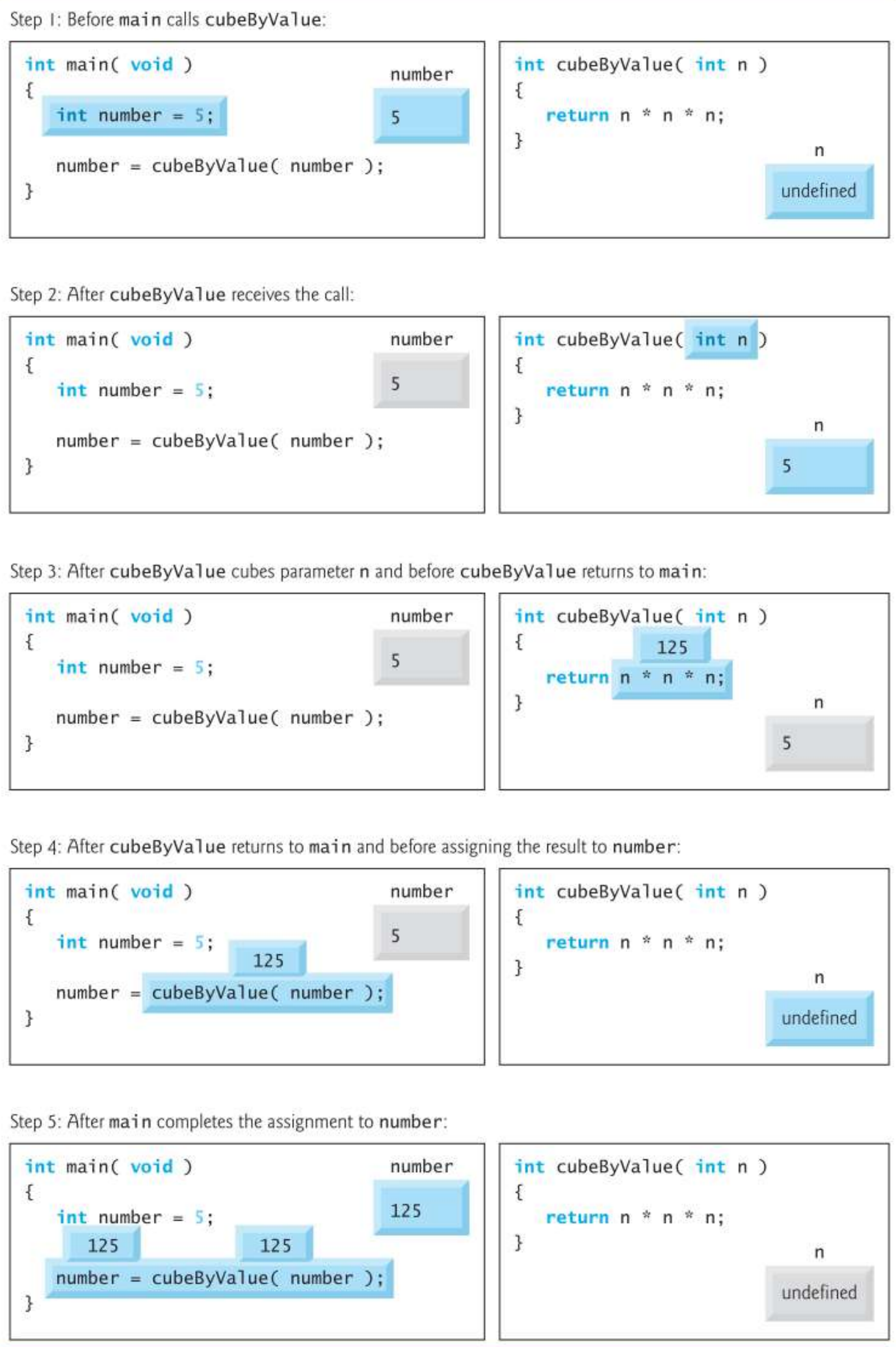
The **`const` qualifier** enables you to inform the compiler that the value of a particular variable should not be modified.

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 improper 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.



Step 1: Before main calls cubeByValue:

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

   number = cubeByValue( number );
}
```

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

Step 2: After cubeByValue receives the call:

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

   number = cubeByValue( number );
}
```

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

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

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

   number = cubeByValue( number );
}
```

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

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

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

   number = cubeByValue( number );
}
```

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

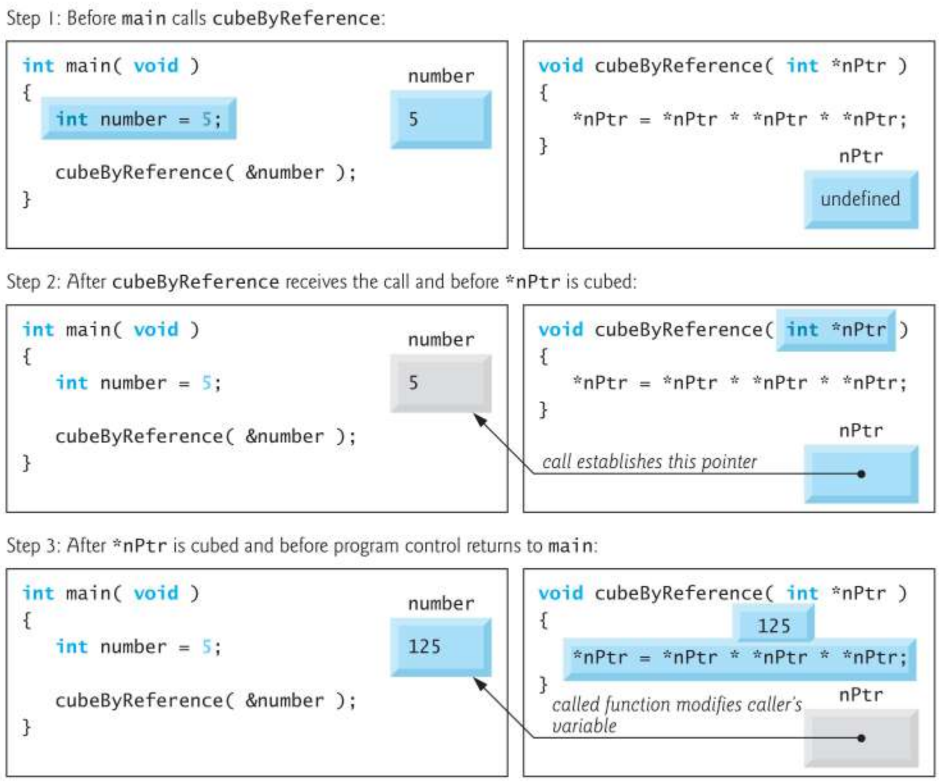
Step 5: After main completes the assignment to number:

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

   number = cubeByValue( number );
}
```

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

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



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

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

   cubeByReference( &number );
}
```

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

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

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

   cubeByReference( &number );
}
```

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

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

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

   cubeByReference( &number );
}
```

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

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

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 single-subscripted array and its size as arguments and prints the array. Such a function should loop through the array and output each array element individually. The size of the array is used in the function body to determine the high subscript of the array, so the loop can terminate when the printing is completed. 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**, and a **constant pointer to constant data**. Each of the four combinations provides different access privileges. These are discussed in the next several examples.

7.5.1 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.

```
 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  } // end main
17
18  // convert string to uppercase letters
19  void convertToUppercase( char *sPtr )
20  {
```

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

```
   while ( *sPtr != '\0' ) { // current character is not '\0'
      *sPtr = toupper( *sPtr ); // convert to uppercase
      ++sPtr; // make sPtr point to the next character
   } // end while
} // end function convertToUppercase
```

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

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

7.5.2 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 the data. 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 to point to the next character in the string.

```
1  // Fig. 7.11: fig07_11.c
2  // Printing a string one character at a time using
3  // a non-constant pointer to constant data.
4
5  #include <stdio.h>
6
7  void printCharacters( const char *sPtr );
8
9  int main( void )
10 {
11    // initialize char array
12    char string[] = "print characters of a string";
13
14    puts( "The string is:" );
15    printCharacters( string );
16    puts( "" );
17 } // end main
18
19 // sPtr cannot modify the character to which it points,
20 // i.e., sPtr is a "read-only" pointer
21 void printCharacters( const char *sPtr )
22 {
```

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

```
   // loop through entire string
   for ( ; *sPtr != '\0'; ++sPtr ) { // no initialization
      printf( "%c", *sPtr );
   } // end for
} // end function printCharacters
```

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

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

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 actual error message you see will be compiler specific.

```
 1  // Fig. 7.12: fig07_12.c
 2  // Attempting to modify data through a
 3  // non-constant pointer to constant data.
 4  #include <stdio.h>
 5  void f( const int *xPtr ); // prototype
 6
 7  int main( void )
 8  {
 9     int y; // define y
10
11     f( &y ); // f attempts illegal modification
12  } // end main
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  } // end function f
```

```
c:\examples\ch07\fig07_12.c(18) : error C2166: l-value specifies const object
```

Fig. 7.12 | Attempting to modify data through a non-constant pointer to constant data.

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** in other languages). A structure is capable of storing related data items of *different* data types under one name (e.g., storing information about each employee of a company). When a function is called with an array as an argument, the array is automatically passed to the function *by reference*. However, structures are always passed *by value*—a *copy* of the entire structure is passed. 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

Pass large objects such as structures using pointers to constant data to obtain 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 subscripting. A constant pointer to non-constant data can be used to receive an array as an argument to a function that accesses array elements using only array subscript notation. Pointers that are declared `const` must be initialized when they'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  } // end main
```

Fig. 7.13 | Attempting to modify a constant pointer to non-constant data. (Part 1 of 2.)

```
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. (Part 2 of 2.)

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 subscript 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
 5  int main( void )
 6  {
 7     int x = 5; // initialize x
 8     int y; // define y
 9
10     // ptr is a constant pointer to a constant integer. ptr always
11     // points to the same location; the integer at that location
12     // cannot be modified
13     const int *const ptr = &x; // initialization is OK
14
15     printf( "%d\n", *ptr );
16     *ptr = 7; // error: *ptr is const; cannot assign new value
17     ptr = &y; // error: ptr is const; cannot assign new address
18  } // end main
```

```
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.

7.6 Bubble Sort Using Pass-by-Reference

Let's improve the bubble sort program of Fig. 6.15 to use two functions—`bubbleSort` and `swap`. Function `bubbleSort` sorts the array. It calls function `swap` (line 50) to exchange the array elements `array[j]` and `array[j + 1]` (Fig. 7.15). Remember that C enforces *information hiding* between functions, so `swap` does not have access to individual array elements in `bubbleSort`. Because `bubbleSort` *wants* `swap` to have access to the array elements to be swapped, `bubbleSort` passes each of these elements *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 50) to effect pass-by-reference as follows

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

Function `swap` receives `&array[j]` in pointer variable `element1Ptr` (line 58). Even though `swap`—because of information hiding—is *not* allowed to know the name `array[j]`, `swap` may use `*element1Ptr` as a *synonym* for `array[j]`—when `swap` references `*element1Ptr`, it's *actually* referencing `array[j]` in `bubbleSort`. Similarly, when `swap` references `*element2Ptr`, it's *actually* referencing `array[j + 1]` in `bubbleSort`. Even though `swap` is not allowed to say

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

precisely the *same* effect is achieved by lines 60 through 62

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

```
1  // Fig. 7.15: fig07_15.c
2  // Putting values into an array, sorting the values into
3  // ascending order and printing the resulting array.
4  #include <stdio.h>
5  #define SIZE 10
6
7  void bubbleSort( int * const array, size_t size ); // prototype
8
9  int main( void )
10 {
11    // initialize array a
12    int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
13
14    size_t i; // counter
15
16    puts( "Data items in original order" );
17
18    // loop through array a
19    for ( i = 0; i < SIZE; ++i ) {
20       printf( "%4d", a[ i ] );
21    } // end for
22
23    bubbleSort( a, SIZE ); // sort the array
24
25    puts( "\nData items in ascending order" );
26
27    // loop through array a
28    for ( i = 0; i < SIZE; ++i ) {
29       printf( "%4d", a[ i ] );
30    } // end for
```

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

```
31
32      puts( "" );
33   } // end main
34
35   // sort an array of integers using bubble sort algorithm
36   void bubbleSort( int * const array, size_t size )
37   {
38      void swap( int *element1Ptr, int *element2Ptr ); // prototype
39      unsigned int pass; // pass counter
40      size_t j; // comparison counter
41
42      // loop to control passes
43      for ( pass = 0; pass < size - 1; ++pass ) {
44
45         // loop to control comparisons during each pass
46         for ( j = 0; j < size - 1; ++j ) {
47
48            // swap adjacent elements if they're out of order
49            if ( array[ j ] > array[ j + 1 ] ) {
50               swap( &array[ j ], &array[ j + 1 ] );
51            } // end if
52         } // end inner for
53      } // end outer for
54   } // end function bubbleSort
55
56   // swap values at memory locations to which element1Ptr and
57   // element2Ptr point
58   void swap( int *element1Ptr, int *element2Ptr )
59   {
60      int hold = *element1Ptr;
61      *element1Ptr = *element2Ptr;
62      *element2Ptr = hold;
63   } // end function swap
```

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

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

Several features of function `bubbleSort` should be noted. The function header (line 36) declares `array` as `int * const array` rather than `int array[]` to indicate that `bubbleSort` receives a single-subscripted array as an argument (again, these notations are interchangeable). Parameter `size` is declared `const` to enforce the principle of least privilege. Although parameter `size` receives a copy of a value in `main`, and modifying the copy cannot change the value in `main`, `bubbleSort` does *not* need to alter `size` to accomplish its task. 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.

The prototype for function `swap` (line 38) 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. Other functions that attempt to call swap do *not* have access to a proper function prototype, so the compiler generates one automatically. This normally results in a prototype that does *not* match the function header (and generates a compilation warning or error) because the compiler assumes int for the return type and the parameter types.

Software Engineering Observation 7.2

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 receives the size of the array as a parameter (line 36). 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 single-subscripted 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 single-subscripted integer arrays of the specific size coded into the function can use the function.

7.7 sizeof Operator

C provides the special unary operator **sizeof** to determine the size in bytes of an array (or any other data type). 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. 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.

```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 ) );
} // end main

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

```
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. The size of a pointer on our system is 4 bytes, so `getSize` returned 4. 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.*

```c
// Fig. 7.17: fig07_17.c
// Using operator sizeof to determine standard data type sizes.
#include <stdio.h>

int main( void )
{
   char c;
   short s;
   int i;
   long l;
   long long ll;
   float f;
   double d;
   long double ld;
   int array[ 20 ]; // create array of 20 int elements
   int *ptr = array; // create pointer to array

   printf( "     sizeof c = %u\tsizeof(char)  = %u"
           "\n     sizeof s = %u\tsizeof(short) = %u"
           "\n     sizeof i = %u\tsizeof(int) = %u"
           "\n     sizeof l = %u\tsizeof(long) = %u"
           "\n    sizeof ll = %u\tsizeof(long long) = %u"
           "\n     sizeof f = %u\tsizeof(float) = %u"
           "\n     sizeof d = %u\tsizeof(double) = %u"
           "\n    sizeof ld = %u\tsizeof(long double) = %u"
           "\n sizeof array = %u"
           "\n   sizeof ptr = %u\n",
          sizeof c, sizeof( char ), sizeof s, sizeof( short ), sizeof i,
          sizeof( int ), sizeof l, sizeof( long ), sizeof ll,
          sizeof( long long ), sizeof f, sizeof( float ), sizeof d,
          sizeof( double ), sizeof ld, sizeof( long double ),
          sizeof array, sizeof ptr );
} // end main
```

```
     sizeof c = 1       sizeof(char)  = 1
     sizeof s = 2       sizeof(short) = 2
     sizeof i = 4       sizeof(int) = 4
     sizeof l = 4       sizeof(long) = 4
    sizeof 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.

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

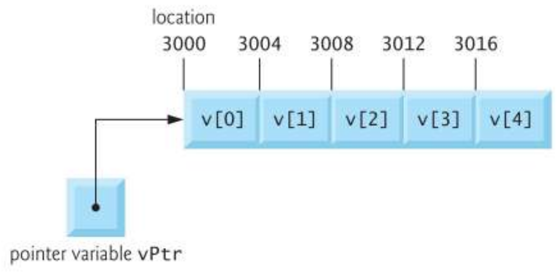


Fig. 7.18 | Array `v` and a pointer variable `vPtr` that points to `v`.

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.

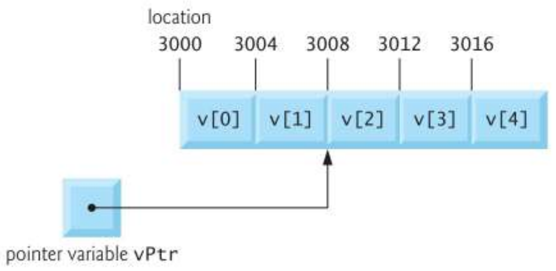


Fig. 7.19 | The pointer vPtr after pointer arithmetic.

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

```
vPtr -= 4;
```

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

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 7.4

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

Common Programming Error 7.5

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

Common Programming Error 7.6

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

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

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

Common Programming Error 7.7

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.

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

7.9 Relationship between Pointers and Arrays

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

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

```
bPtr = b;
```

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

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

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

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

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

Pointers can be subscripted 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/subscript notation**.

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

```
b += 3
```

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

Common Programming Error 7.9

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

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

```
// Fig. 7.20: fig07_20.cpp
// Using subscripting and pointer notations with arrays.
#include <stdio.h>
#define ARRAY_SIZE 4

int main( void )
{
```

Fig. 7.20 | Using subscripting and pointer notations with arrays. (Part 1 of 3.)

```
   int b[] = { 10, 20, 30, 40 }; // create and initialize array b
   int *bPtr = b; // create bPtr and point it to array b
   size_t i; // counter
   size_t offset; // counter

   // output array b using array subscript notation
   puts( "Array b printed with:\nArray subscript notation" );

   // loop through array b
   for ( i = 0; i < ARRAY_SIZE; ++i ) {
      printf( "b[ %u ] = %d\n", i, b[ i ] );
   } // end for

   // output array b using array name and pointer/offset notation
   puts( "\nPointer/offset notation where\n"
         "the pointer is the array name" );

   // loop through array b
   for ( offset = 0; offset < ARRAY_SIZE; ++offset ) {
      printf( "*( b + %u ) = %d\n", offset, *( b + offset ) );
   } // end for

   // output array b using bPtr and array subscript notation
   puts( "\nPointer subscript notation" );

   // loop through array b
   for ( i = 0; i < ARRAY_SIZE; ++i ) {
      printf( "bPtr[ %u ] = %d\n", i, bPtr[ i ] );
   } // end for

   // output array b using bPtr and pointer/offset notation
   puts( "\nPointer/offset notation" );

   // loop through array b
   for ( offset = 0; offset < ARRAY_SIZE; ++offset ) {
      printf( "*( bPtr + %u ) = %d\n", offset, *( bPtr + offset ) );
   } // end for
} // end main
```

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

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

Fig. 7.20 | Using subscripting and pointer notations with arrays. (Part 2 of 3.)

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

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

Fig. 7.20 | Using subscripting and pointer notations with arrays. (Part 3 of 3.)

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.

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

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

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

```
19       copy2( string3, string4 );
20       printf( "string3 = %s\n", string3 );
21    } // end main
22
23    // copy s2 to s1 using array notation
24    void copy1( char * const s1, const char * const s2 )
25    {
26       size_t i; // counter
27
28       // loop through strings
29       for ( i = 0; ( s1[ i ] = s2[ i ] ) != '\0'; ++i ) {
30          ; // do nothing in body
31       } // end for
32    } // end function copy1
33
34    // copy s2 to s1 using pointer notation
35    void copy2( char *s1, const char *s2 )
36    {
37       // loop through strings
38       for ( ; ( *s1 = *s2 ) != '\0'; ++s1, ++s2 ) {
39          ; // do nothing in body
40       } // end for
41    } // end function copy2
```

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

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

Function `copy2` uses *pointers and pointer arithmetic* to copy the string in `s2` to the character array `s1`. Again, the `for` statement header (line 38) performs the entire copy operation. The header does not include any variable initialization. As in function `copy1`, the expression (`*s1 = *s2`) performs the copy operation. Pointer `s2` is dereferenced, and the resulting character is assigned to the dereferenced pointer `*s1`. After the assignment in the condition, the pointers are incremented to point to the next element of array `s1` and the next character of string `s2`, respectively. When the null character is encountered in `s2`, it's assigned to the dereferenced pointer `s1` and the loop terminates.

The first argument to both `copy1` *and* `copy2` *must be an array large enough to hold the string in the second argument.* Otherwise, an error may occur when an attempt is made to write into a memory location that's not part of the array. Also, the second parameter of each function is declared as `const char *` (a constant string). In both functions, the second argument is copied into the first argument—characters are read from it one at a time, but the characters are *never modified*. Therefore, the second parameter is declared to point to a constant value so that the *principle of least privilege* is enforced—neither function requires the capability of modifying the second argument, so neither function is provided with that capability.

7.10 Arrays of Pointers

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

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

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

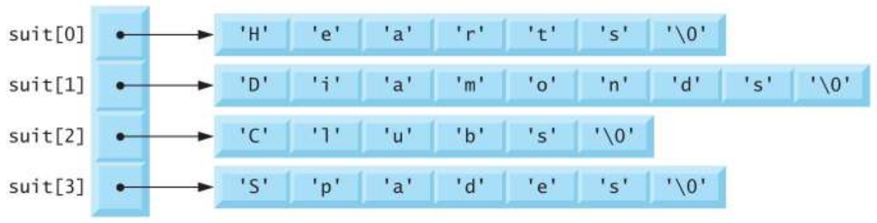

suit[0]
suit[1]
suit[2]
suit[3]
'H' 'e' 'a' 'r' 't' 's' '\0'
'D' 'i' 'a' 'm' 'o' 'n' 'd' 's' '\0'
'C' 'l' 'u' 'b' 's' '\0'
'S' 'p' 'a' 'd' 'e' 's' '\0'

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

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