Chapter 3
Pointers

Specific Learning Outcomes

The students must have the ability to:

1. Understand pointer and its functions;
2. Differentiate variables from pointers;
3. Learn the pointers and pointer operators;
4. Utilize pointers to pass arguments to functions by reference;
5. Understand the close relationships among pointers, arrays and strings;
6. Utilize pointers to functions.

Learning Content

1. Pointers;
2. Variables and Pointers (Similarities and Differences);
3. Using Pointers in Storage of Values;
4. Calling Function by Reference;
5. Strings, Arrays and Pointers
   - Manipulating String Characters through the Use of Pointers
   - Manipulating Array Contents through the Use of Pointers

One of the most powerful features of the C programming language is the pointer. 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.

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 (Figure 3.1). Referencing a value through a pointer is called indirection.

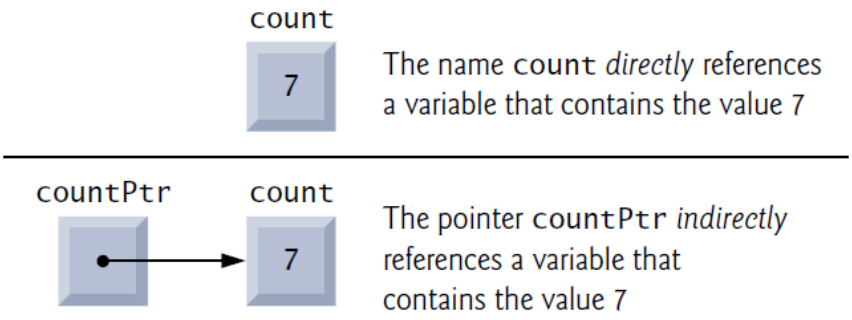


Figure 3.1

Directly and indirectly referencing a variable

Declaring Pointers

Pointers, like all variables, must be defined before they can be used. The definition 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 object2 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. 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. 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 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.

Error-Prevention Tip. Initialize pointers to prevent unexpected results.

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 2 shows a schematic representation of memory after the preceding assignment is executed.

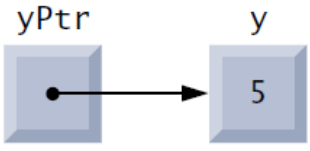


Figure 3.2

Graphical representation of a pointer pointing to an integer variable in memory

Pointer Representation in Memory

Figure 3.3 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.

Figure 3.3

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. 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 executiontime error, or it could accidentally modify important data and allow the program to run to completion with incorrect results.

Demonstrating the & and * Operators

Program 19 demonstrates the pointer operators & and *. The printf conversion specifier %p outputs the memory location as a hexadecimal integer on most platforms. 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 5 lists the precedence and associativity of the operators introduced to this point.

```
1 // Program_19.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", , );
12
13        printf("\n\nThe value of a is %d"
14                "\nThe value of *aPtr is %d", a, );
15
16        printf("\n\nShowing that * and & are complements of "
17                "each other\n&*aPtr = %p"
18                "\n*&aPtr = %p\n", , );
19 }
```

output

```
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
```

Program 19 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

Figure 3.4

Precedence and associativity of the operators discussed so far

Passing Arguments to Functions by Reference

There are two ways to pass arguments to a function—pass-by-value and pass-by- eference. 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). 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. 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 20 and 21 present two versions of a function that cubes an integer, cubeByValue and cubeByReference. Line 14 of program 20 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).

```
1 // Program_20.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
13      // pass number by value to cubeByValue
14      number = cubeByValue(number);
15
16      printf("\nThe new value of number is %d\n", number);
17 }
18
19 // calculate and return cube of integer argument
20 int cubeByValue(int n)
21 {
22      return n * n * n; // cube local variable n and return result
23 }
```

output

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

Program 20 Cube a variable using pass-by-value

Pass-By-Reference

Program 21 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 3.5 and 3.6 analyze graphically and step-by-step the programs in Programs 20 and 21, respectively.

```
1 // Program_21.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
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 }
```

output

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

Program 21 Cube a variable using pass-by-reference with a pointer argument

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 program 21the 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

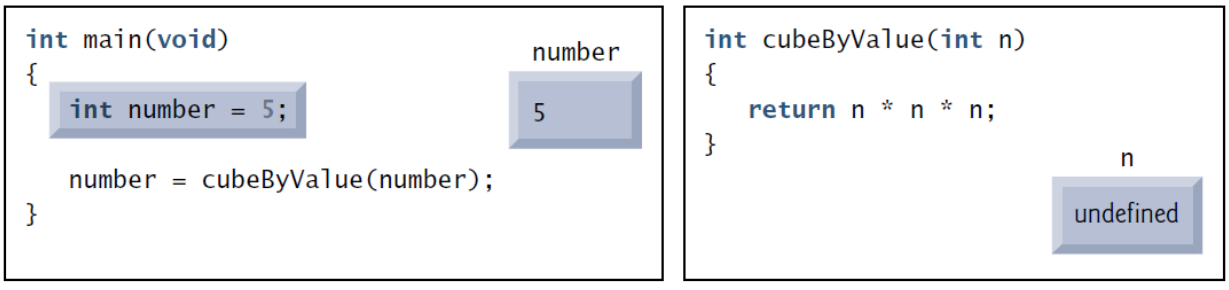
The function prototype for cubeByReference (program 21, line 6) specifies an int * parameter. As with other variable types, it's not necessary to include names of pointers in function prototypes. Names included for documentation purposes are ignored by the C compiler.

Functions That Receive One-Dimensional Arrays

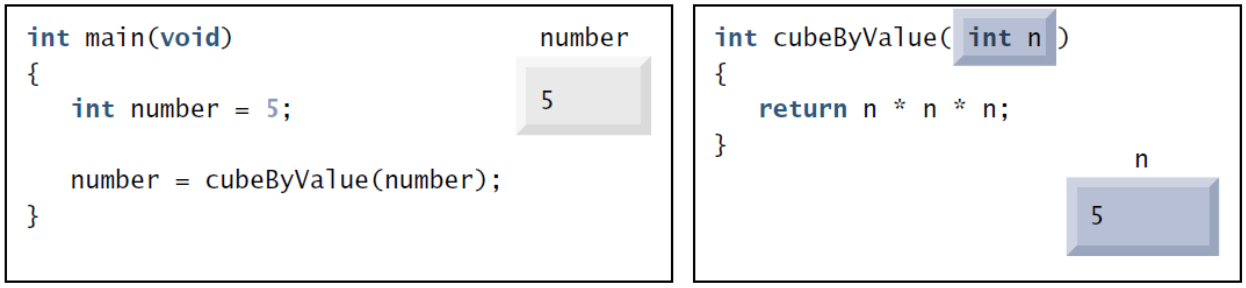
For a function that expects a one-dimensional array as an argument, the function's prototype and header can use the pointer notation shown in the parameter list of function cubeByReference (line 21). The compiler does not differentiate between a function that receives a pointer and one that receives 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. 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.

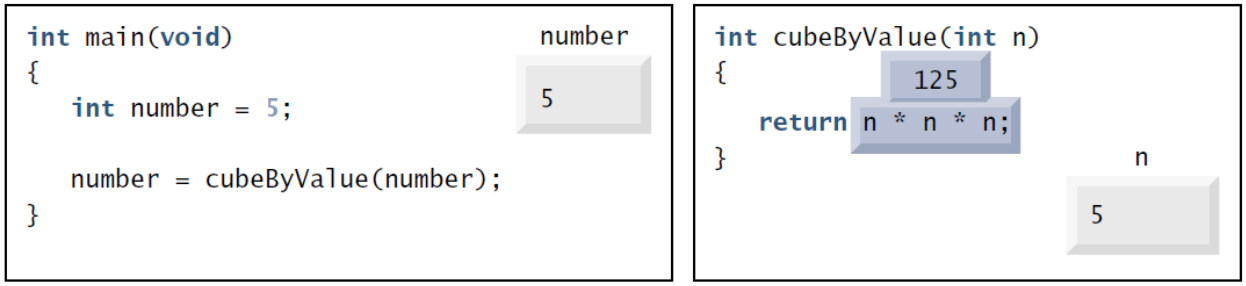
Step 1: Before `main` calls `cubeByValue`:



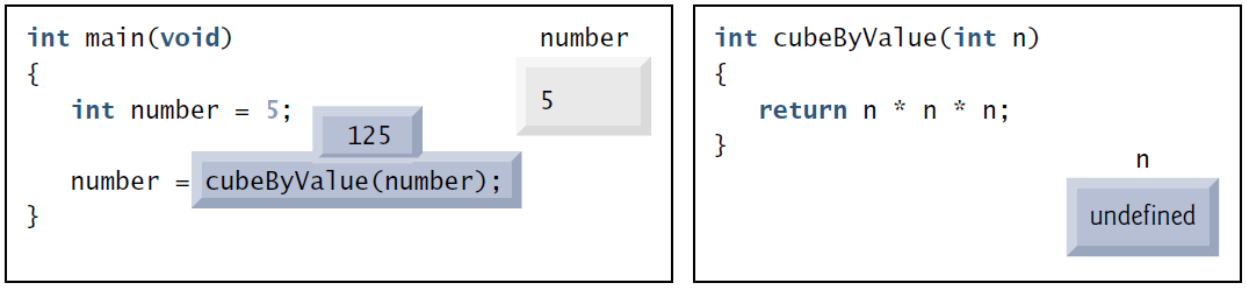
Step 2: After cubeByValue receives the call:



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:

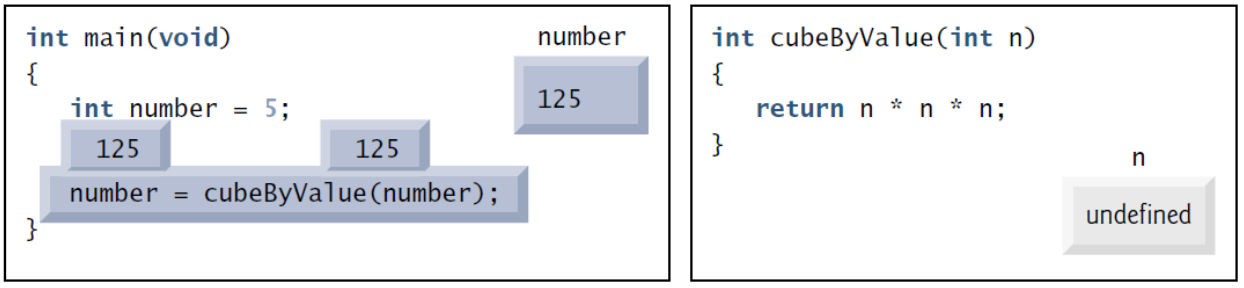


Figure 3.5

Analysis of a typical pass-by-value

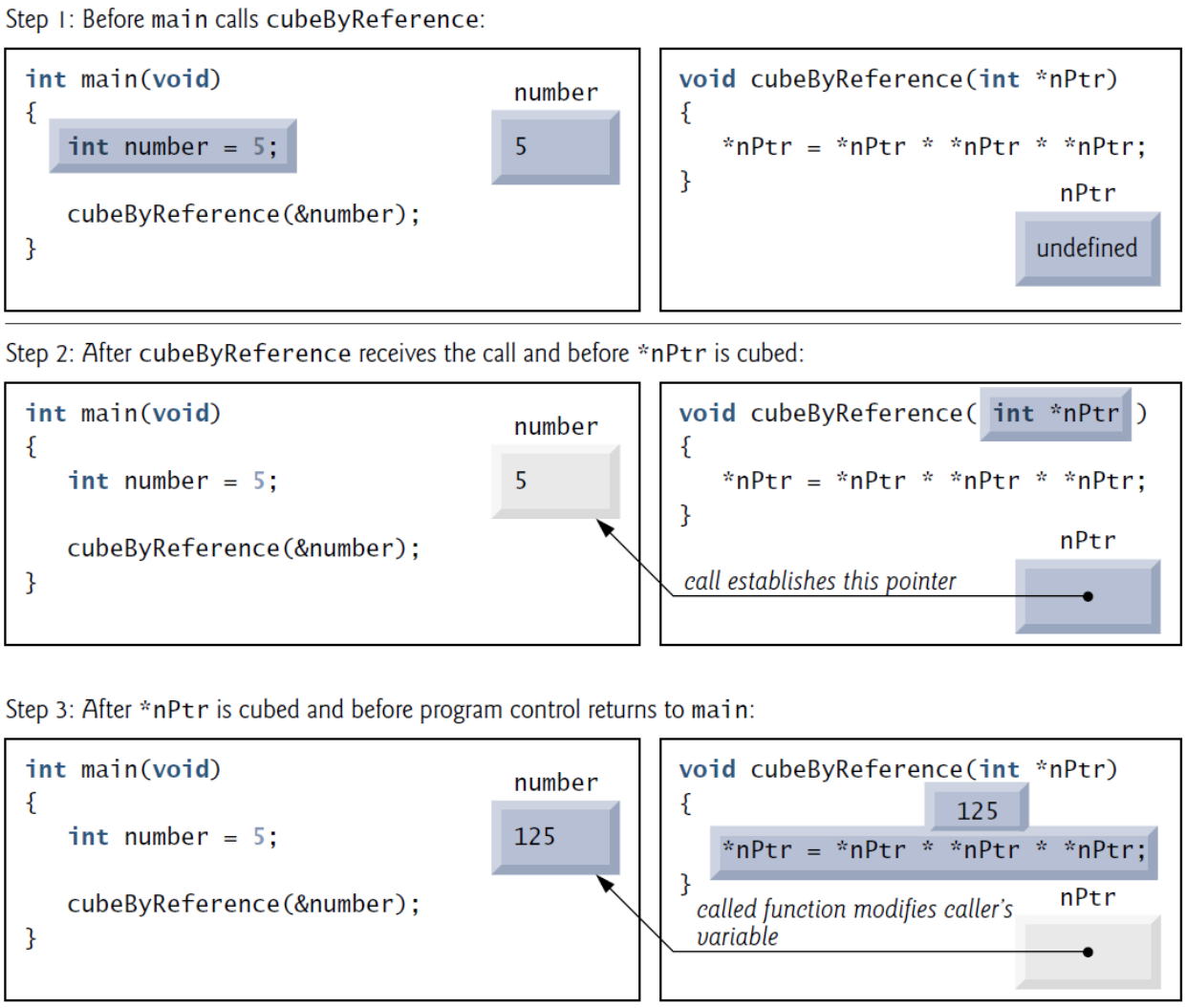


Figure 3.6

Analysis of a typical pass-by-reference with a pointer argument

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. 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 C functions, 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. 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. 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:

1. a non-constant pointer to non-constant data.
2. a constant pointer to nonconstant data.
3. a non-constant pointer to constant data.
4. a constant pointer to constant data.

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

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 Program 22 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. C Characters and Strings presents many C standard library character- and string-processing functions.

```
1 // Program_22.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( ); // 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( )
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 }
```

output

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

Program 22 Converting a string to uppercase using a non-constant pointer to non-constant data

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 (Program 23) 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.

```
1 // Program_23.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( );
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 }
18
19 // sPtr cannot be used to modify the character to which it points,
20 // i.e., sPtr is a "read-only" pointer
21 void printCharacters(const char *sPtr)
22 {
23      // loop through entire string
24      for (; *sPtr != '\0'; ++sPtr) { // no initialization
25              printf("%c", *sPtr);
26      }
27 }
```


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

Program 23 Printing a string one character at a time using a non-constant pointer to constant data

Program 24 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'

```
1 // Program_24.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 }
13
14 // xPtr cannot be used to modify the
15 // value of the variable to which it points
16 void f( )
17 {
18      *xPtr = 100; // error: cannot modify a const object
19 }
```

output

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

Program 24 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 the succeeding chapter, we will 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 ith 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 passby- 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.

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 passby- value is still the best way to prevent data from being modified.

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). Program 25 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 // Program_25.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
```

Program 25 Attempting to modify a constant pointer to non-constant data

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. Program 26 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 // Program_26.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 }
```


```
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
```

Program 26 Attempting to modify a constant pointer to constant data

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. When applied to the name of an array as in Program 27 (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.

```
1 // Program_27.c
2 // Applying sizeof to an array name returns
3 // the number of bytes in the array.
4 #include <stdio.h>
5 #define SIZE 20
6
7 size_t getSize(float *ptr); // prototype
8
9 int main(void)
10 {
11      float array[SIZE]; // create array
12
13      printf("The number of bytes in the array is %u"
14             "\nThe number of bytes returned by getSize is %u\n",
15             sizeof(array) , getSize(array) );
16 }
17
18 // return size of ptr
19 size_t getSize(float *ptr)
20 {
21      return sizeof(ptr);
22 }
```

output

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

Program 27 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:

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

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

sizeof(real) / sizeof(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

Program 28 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 // Program_28.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 }
```

output

```
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
```

Program 28 Using operator sizeof to determine standard data type sizes

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.

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.

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.

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];
```

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

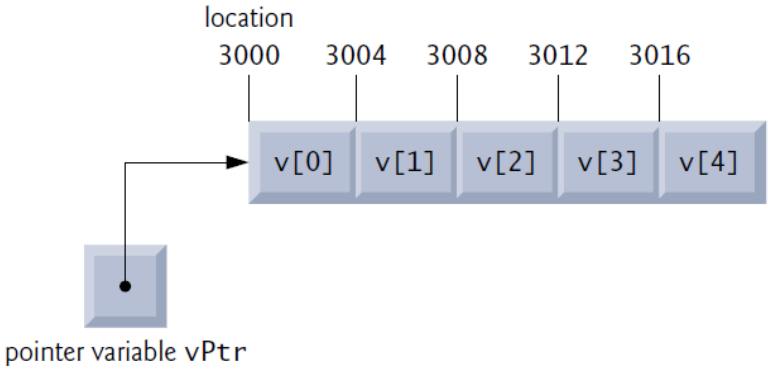


Figure 3.7

Array v and a pointer variable vPtr that points to v

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] (Figure 3.8). 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. Using pointer arithmetic on a pointer that does not refer to an element in an array.

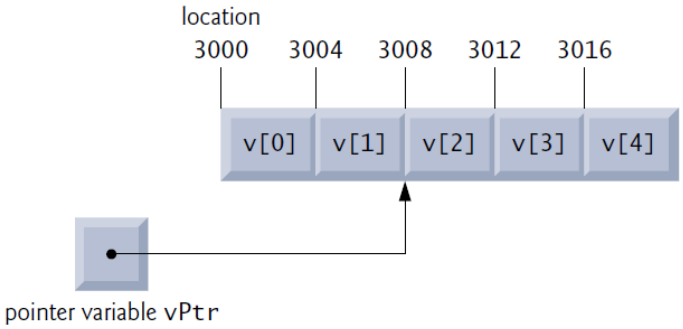


Figure 3.8

The pointer vPtr after pointer arithmetic

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

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.

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

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.

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

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

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];
```

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.

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.

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

Demonstrating Pointer Indexing and Offsets

Program 29 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 // Program_29.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, );
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, *( ));
26      }
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, );
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, *( ));
42      }
43 }
```

output

```
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
```

Program 29 Using indexing and pointer notations with arrays

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 Program 30. 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.

```
1 // Program_30.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
14      copy1(string1, string2);
15      printf("string1 = %s\n", string1);
16
17      char string3[SIZE]; // create array string3
18      char string4[] = "Good Bye"; // create an array containing a string
19
20      copy2(string3, string4);
21      printf("string3 = %s\n", string3);
22 }
23
24 // copy s2 to s1 using array notation
25 void copy1(char * const s1, const char * const s2)
26 {
27      // loop through strings
28      for (size_t i = 0; (s1[i] = s2[i]) != '\0'; ++i) {
29           ; // do nothing in body
30      }
31 }
32
33 // copy s2 to s1 using pointer notation
34 void copy2(char *s1, const char *s2)
35 {
36      // loop through strings
37      for (; (*s1 = *s2) != '\0'; ++s1, ++s2) {
38           ; // do nothing in body
39      }
40 }
```

output

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

Program 30 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.

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 (Figure 3.9). 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 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.

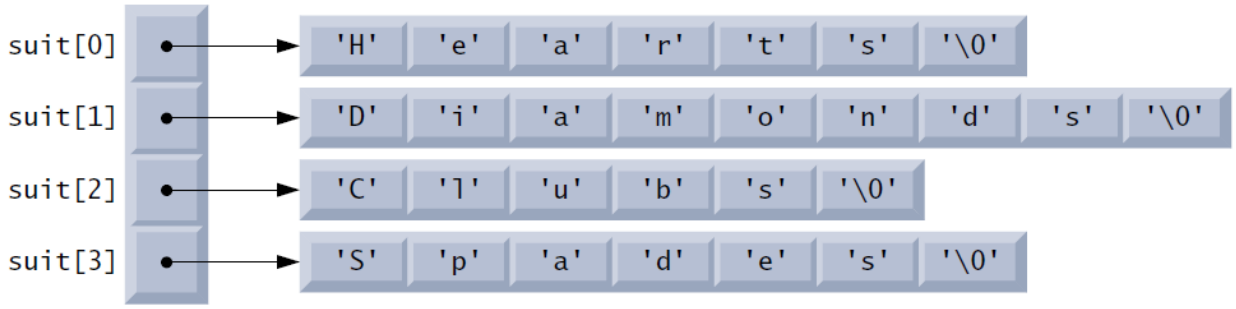


Figure 3.9

Graphical representation of the suit array

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, 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 (Figure 3.10). 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 rows and columns are repeatedly chosen at random until an unselected card is found. Eventually, the numbers 1 through 52 will occupy the 52 slots of the deck array. At this point, the deck of cards is fully shuffled.

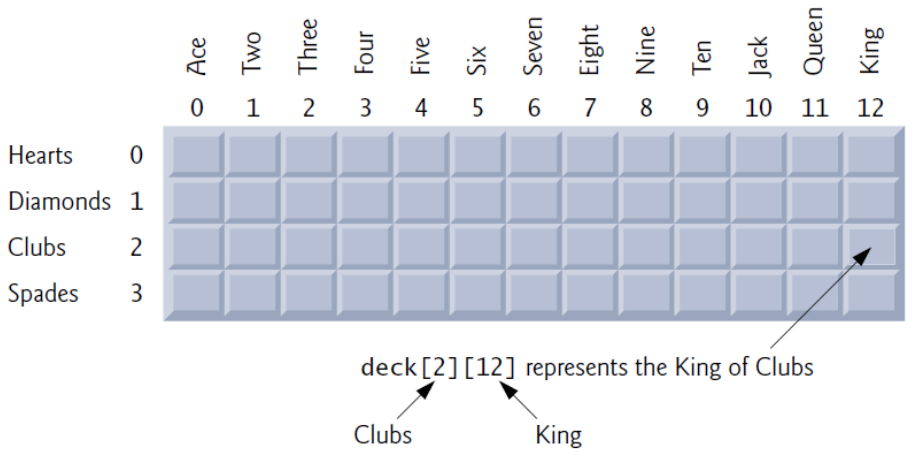


Figure 3.10

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.

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

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, dealing program and a sample execution is shown in Program 31. 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.

```
1 // Program_31.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
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 }
```

output

```
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
```

output

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

Program 31 Card shuffling and dealing

References

Paul Deitel, and Harvey Deitel, “C How to Program with an introduction to C++” Eight Edition

Kimberly Nelson King, “C Programming: A Modern Approach” Second Edition

Brian W. Kernighan and Dennis M. Ritchie, “C Programming Language” 2nd Edition

Jeri R. Hanly and Elliot B. Koffman, “Problem Solving and Program Design in C”, 8th

https://www.thecrazyprogrammer.com

https://www.geeksforgeeks.org

Additional Resources and Links

Introduction to Pointers in C by Neso Academy
https://www.youtube.com/watch?v=f2i0CnUOniA&list=PLBlnK6fEyqRhX6r2uhhlubuF5QextdCSM&index=102&t=12s

Declaring and Initializing Pointers in C by Neso Academy
https://www.youtube.com/watch?v=b3G9RjG4l2s&list=PLBlnK6fEyqRhX6r2uhhlubuF5QextdCSM&index=103

Value of Operator in Pointers C by Neso Academy
https://www.youtube.com/watch?v=xlt_bEqfnxg&list=PLBlnK6fEyqRhX6r2uhhlubuF5QextdCSM&index=104

Pointer Assignment C by Neso Academy
https://www.youtube.com/watch?v=qG01z8unrU4&list=PLBlnK6fEyqRhX6r2uhhlubuF5QextdCSM&index=105

Pointer Application (Finding the Largest & Smallest Elements in an Array) by Neso Academy https://www.youtube.com/watch?v=xlt_bEqfnxg&list= PLBlnK6fEyqRhX6r2uhhlubu F5QextdCSM&index=106

Returning Pointers by Neso Academy https://www.youtube.com/watch?v=xlt_ bEqfnxg&list=PLBlnK6fEyq RhX6r2uhhlubuF5QextdCSM&index=107

Pointers (Important Questions) by Neso Academy https://www.youtube.com/ watch?v=xlt_bEqfnxg&list=PLBlnK6fEyq RhX6r2uhhlubuF5QextdCSM&index=108

Pointer Arithmetic (addition) by Neso Academy https://www.youtube.com/ watch?v=xlt_bEqfnxg&list=PLBlnK6fEyqRhX6r2uhhlubuF5QextdCSM&index=109

Pointer Arithmetic (subtraction) by Neso Academy https://www.youtube.com/ watch?v=xlt_bEqfnxg&list=PLBlnK6fEyqRhX6r2uhhlubuF5QextdCSM&index=110

Pointer Arithmetic (increment and decrement) by Neso Academy https://www.youtube.com/watch?v=xlt_bEqfnxg&list=PLBlnK6fEyqRhX6r2uhhlubu F5QextdCSM&index=111

Pointer Arithmetic (comparing the pointers) by Neso Academy https://www.youtube.com/ watch?v=xlt_bEqfnxg&list=PLBlnK6fEyq RhX6r2uhhlubuF5QextdCSM&index=112

Using Array Name as a Pointer by Neso Academy https://www.youtube.com/ watch?v=xlt_bEqfnxg&list=PLBlnK6fEyq RhX6r2uhhlubuF5QextdCSM&index=114

Pointers (Program 2) | Reverse a Series of Numbers using Pointersby Neso Academy https://www.youtube.com/watch?v=xlt_bEqfnxg&list=PLBlnK6fEyqRhX6r2uhhlubu F5QextdCSM&index=115

Passing Array as an Argument to a Function by Neso Academy https://www.youtube.com/watch?v=xlt_bEqfnxg&list=PLBlnK6fEyqRhX6r2uhhlubu F5QextdCSM&index=116

Processing the Multidimensional Array Elements (or) Address Arithmetic of Multidimensional Arrays by Neso Academy https://www.youtube.com/ watch?v=xlt_bEqfnxg&list=PLBlnK6fEyqRhX6r2uhhlubuF5QextdCSM&index=118

Pointers (Program 3) by Neso Academy https://www.youtube.com/ watch?v=xlt_bEqfnxg&list=PLBlnK6fEyqRhX6r2uhhlubuF5QextdCSM&index=119

Pointers (Program 4) by Neso Academy *https://www.youtube.com/watch?v=xlt_bEqfnxg&list=PLBlnK6fEyqRhX6r2uhhlubuF5QextdCSM&index=120*

Pointer Pointing to an Entire Array by Neso Academy *https://www.youtube.com/watch?v=xlt_bEqfnxg&list=PLBlnK6fEyqRhX6r2uhhlubuF5QextdCSM&index=121*

Pointer Pointing to an Entire Array(Solved Problem) by Neso Academy *https://www.youtube.com/watch?v=xlt_bEqfnxg&list=PLBlnK6fEyqRhX6r2uhhlubuF5QextdCSM&index=122*

Pointers (Program 5) by Neso Academy *https://www.youtube.com/watch?v=xlt_bEqfnxg&list=PLBlnK6fEyqRhX6r2uhhlubuF5QextdCSM&index=123*

Pointers (Program 6) by Neso Academy *https://www.youtube.com/watch?v=xlt_bEqfnxg&list=PLBlnK6fEyqRhX6r2uhhlubuF5QextdCSM&index=124*

Pointers (Program 7) by Neso Academy *https://www.youtube.com/watch?v=xlt_bEqfnxg&list=PLBlnK6fEyqRhX6r2uhhlubuF5QextdCSM&index=125*

Pointers (Program 8) by Neso Academy *https://www.youtube.com/watch?v=xlt_bEqfnxg&list=PLBlnK6fEyqRhX6r2uhhlubuF5QextdCSM&index=126*

Pointers (Program 9) by Neso Academy *https://www.youtube.com/watch?v=xlt_bEqfnxg&list=PLBlnK6fEyqRhX6r2uhhlubuF5QextdCSM&index=127*

Chapter Assessment

Answer the following.

I. Answer the following questions.

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

II. 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.

III. 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?

IV. 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;

Notes

Notes

Notes