Sams Teach Yourself C in 24 Hours

Hour 16 - Applying Pointers

Think twice and do once.

—Chinese proverb

In Hour 11, "An Introduction to Pointers," you learned the basics of using pointers in C. Because pointers are very useful in programming, it's worth spending another hour to learn more about them. In this lesson, the following topics are discussed:

- Pointer arithmetic
- Passing arrays to functions
- Passing pointers to functions
- Pointing to functions

Pointer Arithmetic

In C, you can move the position of a pointer by adding or subtracting integers to or from the pointer. For example, given a character pointer variable ptr_str, the following expression

```
ptr_str + 1
```

indicates to the compiler to move to the memory location that is one byte away from the current position of ptr_str.

Note that for pointers of different data types, the integers added to or subtracted from the pointers have different scalar sizes. In other words, adding 1 to (or subtracting 1 from) a pointer is not instructing the compiler to add (or subtract) one byte to the address, but to adjust the address so that it skips over one element of the type of the pointer. You'll see more details in the following sections.

The Scalar Size of Pointers

The general format to change the position of a pointer is

```
pointer_name + n
```

Here n is an integer whose value can be either positive or negative. pointer_name is the name of a pointer variable that has the following declaration:

```
data_type_specifier *pointer_name;
```

When the C compiler reads the pointer_name + n expression, it interprets the expression as

```
pointer_name + n * sizeof(data_type_specifier)
```

Note that the sizeof operator is used to obtain the number of bytes that a specified data type can have. Therefore, for the char pointer variable ptr_str, the ptr_str + 1 expression actually means

```
ptr_str + 1 * sizeof(char).
```

Because the size of a character is one byte long, ptr_str + 1 tells the compiler to move to the memory location that is 1 byte after the current location referenced by the pointer.

The program in Listing 16.1 shows how the scalar sizes of different data types affect the offsets added to or subtracted from pointers.

TYPE

Listing 16.1. Moving pointers of different data types.

```
1:  /* 16L01.c: Pointer arithmetic */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     char *ptr_ch;
7:     int *ptr_int;
8:     double *ptr_db;
9:     /* char pointer ptr_ch */
10:    printf("Current position of ptr_ch: 0x%p\n", ptr_ch);
11:    printf("The position after ptr_ch + 1: 0x%p\n", ptr_ch + 1);
12:    printf("The position after ptr_ch + 2: 0x%p\n", ptr_ch + 2);
13:    printf("The position after ptr_ch - 1: 0x%p\n", ptr_ch - 1);

14:    printf("The position after ptr_ch - 2: 0x%p\n", ptr_ch - 2);
15:    /* int pointer ptr_int */
16:    printf("Current position of ptr_int: 0x%p\n", ptr_int);
17:    printf("The position after ptr_int + 1: 0x%p\n", ptr_int + 1);
18:    printf("The position after ptr_int + 2: 0x%p\n", ptr_int + 2);
19:    printf("The position after ptr_int - 1: 0x%p\n", ptr_int - 1);
20:    printf("The position after ptr_int - 2: 0x%p\n", ptr_int - 2);
21:    /* double pointer ptr_ch */
22:    printf("Current position of ptr_db: 0x%p\n", ptr_db);
23:    printf("The position after ptr_db + 1: 0x%p\n", ptr_db + 1);
```

```
24:    printf("The position after ptr_db + 2: 0x%p\n", ptr_db + 2);
25:    printf("The position after ptr_db - 1: 0x%p\n", ptr_db - 1);
26:    printf("The position after ptr_db - 2: 0x%p\n", ptr_db - 2);
27:
28:    return 0;
29: }
```

The following output is obtained by running the executable, 16L01.exe, of the program in Listing 16.1 on my machine. You might get a different address on your computer, but the offsets should remain the same:

OUTPUT

```
C:\app>16L01
Current position of ptr_ch: 0x000B
The position after ptr_ch + 1: 0x000C
The position after ptr_ch + 2: 0x000D
The position after ptr_ch - 1: 0x000A
The position after ptr_ch - 2: 0x0009
Current position of ptr_int: 0x028B
The position after ptr_int + 1: 0x028D
The position after ptr_int + 2: 0x028F
The position after ptr_int - 1: 0x0289
The position after ptr_int - 2: 0x0287
Current position of ptr_db: 0x0128
The position after ptr_db + 1: 0x0130
The position after ptr_db + 2: 0x0138
The position after ptr_db - 1: 0x0120
The position after ptr_db - 2: 0x0118
C:\app>
```

ANALYSIS

As you can see in Listing 16.1, there are three types of pointers—ptr_ch, ptr_int, and ptr_db—declared in lines 6_8. Among them, ptr_ch is a pointer to a character, ptr_int is a pointer to an integer, and ptr_db is a pointer to a double.

The statement in line 10 shows the memory address, 0x000B, contained by the char pointer variable ptr_ch. Lines 11 and 12 display the two addresses, 0x000C and 0x000D, when ptr_ch is added to 1 and 2, respectively. Similarly, lines 13 and 14 give 0x000A and 0x0009 when ptr_ch is moved down to lower memory addresses. Because the size of char is 1 byte, ptr_ch+1 means to move to the memory location that is 1 byte higher than the current memory location, 0x000B, referenced by the pointer ptr_ch.

Line 16 shows the memory location referenced by the int pointer variable ptr_int at 0x028B. Because the size of int is 2 bytes long, the ptr_int+1 expression simply means to move to the memory location that is 2 bytes higher than the current one pointed to by ptr_int. That's exactly what has been printed out in line 17. Likewise, line 18 shows that ptr_int+2 causes the reference to be moved to 0x028F, which is 4 bytes higher than 0x028B. The memory location of 0x0289 is referenced by the ptr_int-1 expression in line 19; 0x0287 is referenced by ptr_int-2 in line 20.

The size of the double data type is 8 bytes long. Therefore, the ptr_db+1 expression is interpreted as the memory address referenced by ptr_db plus 8 bytes—that is, 0x0128+8, which gives 0x0130 in hex format. (See the output made by the statement in line 23.)

Lines 24_26 print out the memory addresses referenced by ptr_db+2, ptr_db-1, and ptr_db-2, respectively, which prove that the compiler has taken the scalar size of double in the pointer arithmetic.

WARNING

Pointers are useful if you use them properly. On the other hand, a pointer can get you into trouble if it contains a wrong value. A common error, for instance, is to assign a right value to a pointer that actually expects a left one. Fortunately, many C compilers can find such an error and issue a warning message.
There is another common error that the compiler does not pick up for you: using uninitialized pointers. For example, the following code has a potential problem:

```
int x, ptr_int;
x = 8;
*ptr_int = x;
```

The problem is that the ptr_int pointer is not initialized; it points to some unknown memory location. Therefore, assigning a value, like 8 in this case, to an unknown memory location is dangerous. It may overwrite some important data that is already saved at the memory location, thus causing a serious problem. The solution is to make sure that a pointer is pointing at a legal and valid memory location before it is used.
You can rewrite this C code to avoid the potential problem like this:

```
int x, ptr_int;
x = 8;
ptr_int = &x;   /* initialize the pointer */
```

Pointer Subtraction

For two pointers of the same type, you can subtract one pointer value from the other. For instance, given two char pointer variables, ptr_str1 and ptr_str2, you can calculate the offset between the two memory locations pointed to by the two pointers like this:

```
ptr_str2 - ptr_str1
```

However, it's illegal in C to subtract one pointer value from another if they do not share the same data type.

Listing 16.2 gives an example of performing subtraction on an int pointer variable.

TYPE

Listing 16.2. Performing subtraction on pointers.

```
1:  /* 16L02.c: Pointer subtraction */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int *ptr_int1, *ptr_int2;
7:
8:     printf("The position of ptr_int1: 0x%p\n", ptr_int1);
9:     ptr_int2 = ptr_int1 + 5;
10:    printf("The position of ptr_int2 = ptr_int1 + 5: 0x%p\n", ptr_int2);
11:    printf("The subtraction of ptr_int2 - ptr_int1: %d\n", ptr_int2 -
               Âptr_int1);
12:    ptr_int2 = ptr_int1 - 5;
13:    printf("The position of ptr_int2 = ptr_int1 - 5: 0x%p\n", ptr_int2);
14:    printf("The subtraction of ptr_int2 - ptr_int1: %d\n", ptr_int2 -
       Âptr_int1);
15:
16:    return 0;
17: }
```

After running the executable (16L02.exe) of the program in Listing 16.2 on my machine, I have the following output shown on the screen:

OUTPUT

```
C:\app>16L02
The position of ptr_int1: 0x0128
The position of ptr_int2 = ptr_int1 + 5: 0x0132
The subtraction of ptr_int2 - ptr_int1: 5
The position of ptr_int2 = ptr_int1 - 5: 0x011E
The subtraction of ptr_int2 - ptr_int1: -5
C:\app>
```

ANALYSIS

The program in Listing 16.2 declares two int pointer variables, ptr_int1 and ptr_int2, in line 6. The statement in line 8 prints out the memory position held by ptr_int1. Line 9 assigns the memory address referenced by ptr_int1+5 to ptr_int2. Then, the content of ptr_int2 is printed out in line 10.

The statement in line 11 shows the difference between the two int pointers—that is, the subtraction of ptr_int2 and ptr_int1. The result is 5.

Line 12 then assigns another memory address, referenced by the ptr_int1-5 expression, to the ptr_int2 pointer. Now, ptr_int2 points to a memory location that is 10 bytes lower than the memory location pointed to by ptr_int1 (see the output made by line 13.) The difference between ptr_int2 and ptr_int1 is obtained by the subtraction of the two pointers, which is -5 as printed out by the statement in line 14.

Pointers and Arrays

As indicated in previous lessons, pointers and arrays have a close relationship. You can access an array through a pointer that contains the start address of the array. The following subsection introduces how to access array elements through pointers.

Accessing Arrays via Pointers

Because an array name that is not followed by a subscript is interpreted as a pointer to the first element of the array, you can assign the start address of the array to a pointer of the same data type; then you can access any element in the array by adding a proper integer to the pointer. The value of the integer is the same as the subscript value of the element that you want to access.

In other words, given an array, array, and a pointer, ptr_array, if array and ptr_array are of the same data type, and ptr_array contains the start address of the array, that is

```
ptr_array = array;
```

then the expression array[n] is equivalent to the expression

```
*(ptr_array + n)
```

Here n is a subscript number in the array.

Listing 16.3 demonstrates how to access arrays and change values of array elements by using pointers.

TYPE

Listing 16.3. Accessing arrays by using pointers.

```
1:  /* 16L03.c: Accessing arrays via pointers */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     char str[] = "It's a string!";
7:     char *ptr_str;
8:     int list[] = {1, 2, 3, 4, 5};
9:     int *ptr_int;
10:
11:    /* access char array */
12:    ptr_str = str;
13:    printf("Before the change, str contains: %s\n", str);
14:    printf("Before the change, str[5] contains: %c\n", str[5]);
```

```
15:    *(ptr_str + 5) = `A';
16:    printf("After the change, str[5] contains: %c\n", str[5]);
17:    printf("After the change, str contains: %s\n", str);
18:    /* access int array */
19:    ptr_int = list;
20:    printf("Before the change, list[2] contains: %d\n", list[2]);
21:    *(ptr_int + 2) = -3;
22:    printf("After the change, list[2] contains: %d\n", list[2]);
23:
24:    return 0;
25: }
```

The following output is displayed on the screen after the executable, 16L03.exe, is created and run from a DOS prompt:

OUTPUT

```
C:\app>16L03
Before the change, str contains: It's a string!
Before the change, str[5] contains: a
After the change, str[5] contains: A
After the change, str contains: It's A string!
Before the change, list[2] contains: 3
After the change, list[2] contains: -3
C:\app>
```

ANALYSIS

The purpose of the program in Listing 16.3 is to access a char array, str, and an int array, list. In lines 6 and 8, str and list are declared and initialized with a string and a set of integers, respectively. A char pointer, ptr_str, and an int pointer, ptr_int, are declared in lines 7 and 9.

Line 12 assigns the start address of the str array to the ptr_str pointer. The statements in lines 13 and 14 demonstrate the content of the string saved in the str array, as well as the character contained by the str[5] element in the array before any changes are made to str.

The statement in line 15 shows that the character constant, `A', is assigned to the element of the str array pointed to by the expression

```
*(ptr_str + 5)
```

To verify that the content of the element in str has been updated, lines 16 and 17 print out the element and the whole string, respectively. The output indicates that `A' has replaced the original character constant, `a'.

The start address of the int array list is assigned to the ptr_int pointer in line 19. Before I do anything with the list[2] element of the list array, I print out its value, which is 3 at this moment (see the output made by line 20). In line 21, the list[2] element is given another value, -3, through the dereferenced pointer, *(ptr_int + 2). The printf() function in line 22 prints the latest value of list[2].

Pointers and Functions

Before I talk about passing pointers to functions, let's first have a look at how to pass arrays to functions.

Passing Arrays to Functions

In practice, it's usually awkward if you pass more than five or six arguments to a function. One way to save the number of arguments passed to a function is to use arrays. You can put all variables of the same type into an array, and then pass the array as a single argument.

The program in Listing 16.4 shows how to pass an array of integers to a function.

TYPE

Listing 16.4. Passing arrays to functions.

```
1:  /* 16L04.c: Passing arrays to functions */
2:  #include <stdio.h>
3:
4:  int AddThree(int list[]);
5:
6:  main()
7:  {
8:     int sum, list[3];
9:
10:    printf("Enter three integers separated by spaces:\n");
11:    scanf("%d%d%d", &list[0], &list[1], &list[2]);
12:    sum = AddThree(list);
13:    printf("The sum of the three integers is: %d\n", sum);
14:
15:    return 0;
16: }
17:
18: int AddThree(int list[])
19: {
20:    int i;
21:    int result = 0;
22:
23:    for (i=0; i<3; i++)
24:       result += list[i];
25:    return result;
26: }
```

The following output is obtained after I run the executable, 16L04.exe, and enter three integers, 10, 20, and 30, from a DOS prompt:

OUTPUT

```
C:\app>16L04
Enter three integers separated by spaces:
10  20  30
The sum of the three integers is: 60
C:\app>
```

ANALYSIS

The purpose of the program in Listing 16.4 is to obtain three integers entered by the user, and then pass the three integers as an array to a function called AddThree() to perform the operation of addition.

Line 4 gives the declaration of the AddThree() function. Note that the unsized array, list[], is used in the argument expression, which indicates that the argument contains the start address of the list array.

The list array and an integer variable, sum, are declared in line 8. The printf() function in line 10 displays a message asking the user to enter three integers. Then, line 11 fetches the integers entered by the user and stores them in the three memory locations of the elements in the integer array referenced by &list[0], &list[1], and &list[2], respectively.

The statement in line 12 calls the AddThree() function with the name of the array as the argument. The AddThree(list) expression is actually passing the start address of the list array to the AddThree() function.

The definition of the AddThree() function is in lines 18_26; it adds the values of all three elements in the list array and returns the sum. The result returned from the AddThree() function is assigned to the integer variable sum in line 12 and is printed out in line 13.

NOTE

You can also specify the size of an array that is passed to a function. For instance, the following

```
function(char str[16]);
```

is equivalent to the following statement:

```
function(char str[]);
```

Remember that the compiler can figure out the size for the unsized array str[].
For multidimensional arrays, the format of an unsized array should always be used in the declaration. (See the section titled "Passing Multidimensional Arrays as Arguments," later in this hour.)

Passing Pointers to Functions

As you know, an array name that is not followed by a subscript is interpreted as a pointer to the first element of the array. In fact, the address of the first element in an array is the start address of the array. Therefore, you can assign the start address of an array to a pointer, and then pass the pointer name, instead of the unsized array, to a function.

Listing 16.5 gives an example of passing pointers to functions, which is similar to the situation in which arrays are passed to functions.

TYPE

Listing 16.5. Passing pointers to functions.

```
1:  /* 16L05.c: Passing pointers to functions */
2:  #include <stdio.h>
3:
4:  void ChPrint(char *ch);
5:  int DataAdd(int *list, int max);
6:  main()
7:  {
8:     char str[] = "It's a string!";
9:     char *ptr_str;
10:    int list[5] = {1, 2, 3, 4, 5};
11:    int *ptr_int;
12:
13:    /* assign address to pointer */
14:    ptr_str = str;
15:    ChPrint(ptr_str);
16:    ChPrint(str);
17:
18:    /* assign address to pointer */
19:    ptr_int = list;
20:    printf("The sum returned by DataAdd(): %d\n",
21:            DataAdd(ptr_int, 5));
22:    printf("The sum returned by DataAdd(): %d\n",
23:            DataAdd(list, 5));
24:    return 0;
25: }
26: /* function definition */
27: void ChPrint(char *ch)
28: {
29:    printf("%s\n", ch);
30: }
31: /* function definition */
32: int DataAdd(int *list, int max)
33: {
34:    int i;
35:    int sum = 0;
36:
37:    for (i=0; i<max; i++)
```

```
38:       sum += list[i];
39:    return sum;
40: }
```

After executing the 16L05.exe program, the following output is displayed on the screen:

OUTPUT

```
C:\app>16L05
It's a string!
It's a string!
The sum returned by DataAdd(): 15
The sum returned by DataAdd(): 15
C:\app>
```

ANALYSIS

The purpose of the program in Listing 16.5 is to demonstrate how to pass an integer pointer that points to an integer array and a character pointer that references a character string to two functions that are declared in lines 4 and 5.

Note that expressions, such as char *ch and int *list, are used as arguments in the function declarations, which indicates to the compiler that a char pointer and an int pointer are respectively passed to the functions ChPrint() and DataAdd().

Inside the main() function body, lines 8 and 9 declare a char array (str) that is initialized with a character string, and a char pointer variable (ptr_str). Line 10 declares and initializes an int array (list) with a set of integers. An int pointer variable, ptr_int, is declared in line 11.

The start address of the str array is assigned to the ptr_str pointer by the assignment statement in line 14. Then, the ptr_str pointer is passed to the ChPrint() function as the argument in line 15. According to the definition of ChPrint() in lines 27_30, the content of the str array whose start address is passed to the function as the argument is printed out by the printf() function that is invoked inside the ChPrint() function in line 29.

In fact, you can still use the name of the str array as the argument and pass it to the ChPrint() function. Line 16 shows that the start address of the character array is passed to ChPrint() via the name of the array.

The statement in line 19 assigns the start address of the integer array list to the integer pointer ptr_int. Then, the ptr_int pointer is passed to the DataAdd() function in line 21, along with 5, which is the maximum number of the elements contained by the list array. From the definition of the DataAdd() function in lines 32_40, you can see that DataAdd() adds all the integer elements in list and returns the sum to the caller. Thereafter, the statement in lines 20 and 21 prints out the result returned from DataAdd().

The expression in line 23 also invokes the DataAdd() function, but this time, the name of the list array is used as the argument to the function. Not surprisingly, the start address of the list array is passed to the DataAdd() function successfully, and the printf() statement in lines 22 and 23 displays the right result on the screen.

Passing Multidimensional Arrays as Arguments

In Hour 12, "Storing Similar Data Items," you learned about multidimensional arrays. In this section, you're going to see how to pass multidimensional arrays to functions.

As you might have guessed, passing a multidimensional array to a function is similar to passing a one-dimensional array to a function. You can either pass the unsized format of a multidimensional array or a pointer that contains the start address of the multidimensional array to a function. Listing 16.6 is an example of these two methods.

TYPE

Listing 16.6. Passing multidimensional arrays to functions.

```
1:  /* 16L06.c: Passing multidimensional arrays to functions */
2:  #include <stdio.h>
3:  /* function declarations */
4:  int DataAdd1(int list[][5], int max1, int max2);
5:  int DataAdd2(int *list, int max1, int max2);
6:  /* main() function */
7:  main()
8:  {
9:     int list[2][5] = {1, 2, 3, 4, 5,
10:                       5, 4, 3, 2, 1};
11:    int *ptr_int;
12:
13:    printf("The sum returned by DataAdd1(): %d\n",
14:            DataAdd1(list, 2, 5));
15:    ptr_int = &list[0][0];
16:    printf("The sum returned by DataAdd2(): %d\n",
17:            DataAdd2(ptr_int, 2, 5));
18:
19:    return 0;
20: }
21: /* function definition */
22: int DataAdd1(int list[][5], int max1, int max2)
23: {
24:    int i, j;
25:    int sum = 0;
26:
27:    for (i=0; i<max1; i++)
28:       for (j=0; j<max2; j++)
29:       sum += list[i][j];
30:    return sum;
31: }
32: /* function definition */
```

```
33: int DataAdd2(int *list, int max1, int max2)
34: {
35:    int i, j;
36:    int sum = 0;
37:
38:    for (i=0; i<max1; i++)
39:       for (j=0; j<max2; j++)
40:       sum += *(list + i*max2 + j);
41:    return sum;
42: }
```

The following output is displayed on the screen after the executable (16L06.exe) is executed:

OUTPUT

```
C:\app>16L06
The sum returned by DataAdd1(): 30
The sum returned by DataAdd2(): 30
C:\app>
```

ANALYSIS

At the beginning of the program in Listing 16.6, I declare two functions, DataAdd1() and DataAdd2(), in lines 4 and 5. Note that the first argument to DataAdd1() in line 4 is the unsized array of list. In fact, list is a two-dimensional integer array declared in lines 9 and 10 inside the main() function body. The other two arguments, max1 and max2, are two dimension sizes of the list array.

As you can tell from the definition of DataAdd1() in lines 22_31, each element of the list array, expressed as list[i][j], is added and assigned to a local variable called sum that is returned at the end of the DataAdd1() function in line 30. Here i is from 0 to max1 - 1, and j is within the range of 0 to max2 - 1.

The DataAdd1() function is called in line 14, with the name of the list array and the two dimension sizes, 2 and 5. The result returned by DataAdd1() is printed out by the statement in lines 13 and 14. So you see, passing a multidimensional array to a function is quite similar to passing a one-dimensional array to a function.

Another way to do the job is to pass a pointer that contains the start address of a multidimensional array to a function. In this example, the DataAdd2() function is declared in line 5 with a pointer expression, int *list, as the function's first argument. The definition of DataAdd2() is given in lines 33_42.

Note that in line 40, each element in the list array is fetched by moving the pointer to point to the memory location of the element. That is, the dereferenced pointer *(list + i*max2 + j) returns the value of an element that is located at row i and column j, if you imagine that the two-dimensional array has both a horizontal and a vertical dimension. Therefore, adding i*max2 to list calculates the address of row i (that is, rows 0 through i-1 are skipped over); then adding j calculates the address of element j (that is, column j) in the current row (i). In this example, the range of the row is from 0 to 1 (that is, 2 rows total); the range of the column is from 0 to 4 (that is, 5 columns total). (See Figure 16.1.)

The result returned by the DataAdd2() function is displayed on the screen by the statement declared in lines 16 and 17.

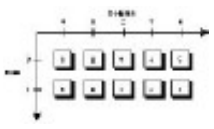

Figure 16.1. *The two-dimensional coordinate shows the locations of the elements in the list array.*

Arrays of Pointers

In many cases, it's useful to declare an array of pointers and access the contents pointed to by the array by dereferencing each pointer. For instance, the following declaration declares an int array of pointers:

```
int *ptr_int[3];
```

In other words, the variable ptr_int is a three-element array of pointers to integers. In addition, you can initialize the array of pointers. For example:

```
int x1 = 10;
int x2 = 100;
int x3 = 1000;
ptr_int[0] = &x1;
ptr_int[1] = &x2;
ptr_int[2] = &x3;
```

Listing 16.7 shows another example. Here an array of pointers is used to access arrays of strings.

TYPE

Listing 16.7. Using an array of pointers to character strings.

```
1:  /* 16L07.c: Using an array of pointers */
2:  #include <stdio.h>
3:  /* function declarations */
4:  void StrPrint1(char **str1, int size);
5:  void StrPrint2(char *str2);
6:  /* main() function */
7:  main()
8:  {
9:     char *str[4] = {"There's music in the sighing of a reed;",
10:                    "There's music in the gushing of a rill;",
```

```
11:                    "There's music in all things if men had ears;",
12:                    "There earth is but an echo of the spheres.\n"
13:                   };
14:    int i, size = 4;
15:
16:    StrPrint1(str, size);
17:    for (i=0; i<size; i++)
18:       StrPrint2(str[i]);
19:
20:    return 0;
21: }
22: /* function definition */
23: void StrPrint1(char **str1, int size)
24: {
25:    int i;
26:    /* Print all strings in an array of pointers to strings */
27:    for (i=0; i<size; i++)
28:       printf("%s\n", str1[i]);
29: }
30: /* function definition */
31: void StrPrint2(char *str2)
32: {
33:     /* Prints one string at a time */
34:    printf("%s\n", str2);
35: }
```

A piece of a poem written by Lord Byron is printed out after the executable (16L07.exe) of the program in Listing 16.7 is created and executed:

OUTPUT

```
C:\app>16L07
There's music in the sighing of a reed;
There's music in the gushing of a rill;
There's music in all things if men had ears;
There earth is but an echo of the spheres.
There's music in the sighing of a reed;
There's music in the gushing of a rill;
There's music in all things if men had ears;
There earth is but an echo of the spheres.
C:\app>
```

ANALYSIS

Let's first have a look at the array of pointers, str, which is declared and initialized in lines 9_13 inside the main() function body of the program in Listing 16.7. As you can see, str is a four-element array of pointers to a set of character strings. I have adopted four sentences of a poem written by Lord Byron and used them as four character strings in the program.

You can get access to a character string by using a corresponding pointer in the array. In fact, there are two functions, StrPrint1() and StrPrint2(), in Listing 16.7. Both of them can be called to gain access to the character strings. From the function declaration in line 4, you can see that the StrPrint1() function is passed with a pointer of pointers—that is, **str1, which is dereferenced inside the StrPrint1() function to represent the four pointers that point to the four character strings. The definition of StrPrint1() is in lines 23_29.

The StrPrint2() function, on the other hand, only takes a pointer variable as its argument, and prints out a character string referenced by the pointer. Lines 31_35 give the definition of the StrPrint2() function.

Now back to the main() function. The StrPrint1() function is called in line 16 with the name of the array of pointers, str, as the argument. StrPrint1() then displays the four sentences of Byron's poem on the screen. The for loop in lines 17 and 18 does the same thing by calling the StrPrint2() function four times. Each time, the start address of a sentence is passed to StrPrint2(). Therefore, you see all the sentences of the poem printed on the screen twice.

Pointing to Functions

Before you finish the course for this hour, there is one more interesting thing you need to learn about: pointers to functions.

As with pointers to arrays, you can declare a pointer that is initialized with the left value of a function. (The left value is the memory address at which the function is located.) Then you can call the function via the pointer.

The program in Listing 16.8 is an example that declares a pointer to a function.

TYPE

Listing 16.8. Pointing to a function.

```
1:  /* 16L08.c: Pointing to a function */
2:  #include <stdio.h>
3:  /* function declaration */
4:  int StrPrint(char *str);
5:  /* main() function */
6:  main()
7:  {
8:     char str[24] = "Pointing to a function.";
9:     int (*ptr)(char *str);
10:
11:    ptr = StrPrint;
12:    if (!(*ptr)(str))
13:       printf("Done!\n");
14:
15:    return 0;
```

```
16: }
17: /* function definition */
18: int StrPrint(char *str)
19: {
20:    printf("%s\n", str);
21:    return 0;
22: }
```

After the executable, 16L08.exe, of the program in Listing 16.8 is created and executed, the following output is shown on the screen:

OUTPUT

```
C:\app>16L08
Pointing to a function.
Done!
C:\app>
```

ANALYSIS

As usual, a function declaration comes first in Listing 16.8. The StrPrint() function is declared with the int data type specifier and an argument of a char pointer in line 4.

The statement in line 9 gives the declaration of a pointer (ptr) to the StrPrint() function (that is, int (*ptr)(char *str);).

Note that the pointer, ptr, is specified with the int data type and passed with a char pointer. In other words, the format of the pointer declaration in line 9 is quite similar to the declaration of StrPrint() in line 4. Please remember that you have to put the *ptr expression between a pair of parentheses ((and)) so that the compiler won't confuse it with a function name.

In line 11, the left value (that is, the address) of the StrPrint() function is assigned to the ptr pointer. Then, the (*ptr)(str) expression in line 12 calls the StrPrint() function via the dereferenced pointer ptr, and passes the address of the string declared in line 8 to the function.

From the definition of the StrPrint() function in lines 18_22, you can tell that the function prints out the content of a string whose address is passed to the function as the argument. Then, 0 is returned at the end of the function.

In fact, the if statement in lines 12 and 13 checks the value returned by the StrPrint() function. When the value is 0, the printf() function in line 13 displays the string of Done! on the screen.

The output of the program in Listing 16.8 shows that the StrPrint() function has been invoked successfully by using a pointer that holds the address of the function.

Summary

In this lesson you've learned the following:

- You should always make sure that a pointer is pointing to a legal and valid memory location before you use it.
- The position of a pointer can be moved by adding or subtracting an integer.
- The scalar size of a pointer is determined by its data type, which is specified in the pointer declaration.
- For two pointers of the same type, you can subtract one pointer value from the other.
- The elements in an array can be accessed via a pointer that holds the start address of the array.
- You can pass an unsized array as a single argument to a function.
- Also, you can pass an array to a function through a pointer. The pointer should hold the start address of the array.
- You can either pass the unsized format of a multidimensional array or a pointer that contains the start address of the multidimensional array to a function.
- Arrays of pointers are useful in many cases that deal with character strings.
- You can call a function via a pointer that holds the address of the function.

In the next lesson you'll learn how to allocate memory in C.

Q&A

Q Why do you need pointer arithmetic?

A The beauty of using pointers is that you can move pointers around to get access to valid data that is saved in those memory locations referenced by the pointers. To do so, you can perform the pointer arithmetic to add (or subtract) an integer to (or from) a pointer. For example, if a character pointer, ptr_str, holds the start address of a character string, the ptr_str+1 expression means to move to the next memory location that contains the second character in the string.

Q How does the compiler determine the scalar size of a pointer?

A The compiler determines the scalar size of a pointer by its data type specified in the declaration. When an integer is added to or subtracted from a pointer, the actual value the compiler uses is the multiplication of the integer and the size of the pointer type. For instance, given an int pointer ptr_int, the ptr_int + 1 expression is interpreted by the compiler as ptr_int + 1 * sizeof(int). If the size of the int type is 2 bytes, then the ptr_int + 1 expression really means to move 2 bytes higher from the memory location referenced by the ptr_int pointer.

Q How do you get access to an element in an array by using a pointer?

A For a one-dimensional array, you can assign the start address of an array to a pointer of the same type, and then move the pointer to the memory location that contains the value of an element in which you're interested. Then you dereference the pointer to obtain the value of the element. For multidimensional arrays, the method is similar, but you have to think about the other dimensions at the same time. (See the example shown in Listing 16.6.)

Q Why do you need to use arrays of pointers?

A In many cases, it's helpful to use arrays of pointers. For instance, it's convenient to use an array of pointers to point to a set of character strings so that you can access any one of the strings referenced by a corresponding pointer in the array.

Workshop

To help solidify your understanding of this hour's lesson, you are encouraged to answer the quiz questions and finish the exercises provided in the Workshop before you move to the next lesson. The answers and hints to the questions and exercises are given in Appendix E, "Answers to Quiz Questions and Exercises."

Quiz

1. Given a char pointer, ptr_ch, an int pointer, ptr_int, and a float pointer, ptr_flt, how many bytes will be added, respectively, in the following expressions on your machine?
   - ptr_ch + 4
   - ptr_int + 2
   - ptr_flt + 1
   - ptr_ch + 12
   - ptr_int + 6
   - ptr_flt + 3

2. If the address held by an int pointer, ptr1, is 0x100A, and the address held by another int pointer, ptr2, is 0x1006, what will you get from the subtraction of ptr1-ptr2?
3. Given that the size of the double data type is 8 bytes long, and the current address held by a double pointer variable, ptr_db, is 0x0238, what are the addresses held, respectively, by ptr_db-1 and ptr_db+5?
4. Given the following declarations and assignments:

```
char ch[] = {`a', `b', `c', `d', `A', `B', `C', `D'};
char *ptr;
ptr = &ch[1];
```

   what do these expressions do separately?

   - *(ptr + 3)
   - ptr - ch
   - *(ptr - 1)
   - *ptr = `F'

Exercises

1. Given a character string, I like C!, write a program to pass the string to a function that displays the string on the screen.
2. Rewrite the program of exercise 1. This time, change the string of I like C! to I love C! by moving a pointer that is initialized with the start address of the string and updating the string with new characters. Then, pass the updated string to the function to display the content of the string on the screen.
3. Given a two-dimensional character array, str, that is initialized as

```
char str[2][15] = { "You know what,", "C is powerful." };
```

   write a program to pass the start address of str to a function that prints out the content of the character array.

4. Rewrite the program in Listing 16.7. This time, the array of pointers is initialized with the following strings:

   `"Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", and "Saturday".`