# Chapter 9 Understanding Pointers

## 9-1. What Is a Pointer?

To understand pointers, you need a basic knowledge of how your computer stores infor-mation in memory. The following is a somewhat simplified account of PC memory storage.

### 9-1-1. Your Computer's Memory

A PC's RAM consists of many thousands of sequential storage locations, and each location is identified by a unique address. The memory addresses in a given computer range from 0 to a maximum value that depends on the amount of memory installed.

When you're using your computer, the operating system uses some of the system's memory. When you're running a program, the program's code (the machine-language instructions for the program's various tasks) and data (the information the program is using) also use some of the system's memory. This section examines the memory storage for program data.

When you declare a variable in a C program, the compiler sets aside a memory location with a unique address to store that variable. The compiler associates that address with the variable's name. When your program uses the variable name, it automatically accesses the proper memory location. The location's address is used, but it is hidden from you, and you need not be concerned with it.


**Figure 9-1. A program variable is stored at a specific memory address.**

Figure 9.1 shows this schematically. A variable named rate has been declared and initialized to 100. The compiler has set aside storage at address 1004 for the variable and has associated the name rate with the address 1004.

### 9-1-2. Creating a Pointer

You should note that the address of the variable rate (or any other variable) is a number and can be treated like any other number in C. If you know a variable's address, you can create a second variable in which to store the address of the first. The first step is to declare a variable to hold the address of rate. Give it the name p_rate, for example. At first, p_rate is uninitialized. Storage has been allocated for p_rate, but its value is undetermined, as shown in Figure 9.2.

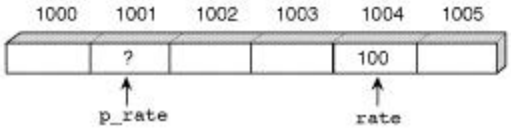


**Figure 9-2. Memory storage space has been allocated for the variable p_rate.**

The next step is to store the address of the variable rate in the variable p_rate. Because p_rate now contains the address of rate, it indicates the location where rate is stored in memory. In C parlance, p_rate points to rate, or is a pointer to rate. This is shown in Figure 9.3.

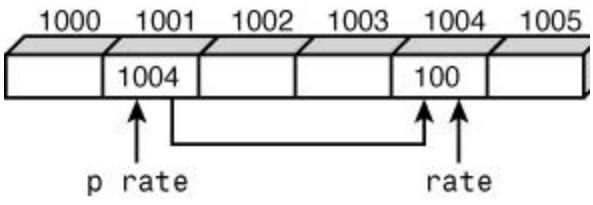


**Figure 9-3. The variable p_rate contains the address of the variable rate and is therefore a pointer to rate.**

To summarize, a pointer is a variable that contains the address of another variable. Now you can get down to the details of using pointers in your C programs.

## 9-2. Pointers and Simple Variables

In the example just given, a pointer variable pointed to a simple (that is, nonarray) variable. This section shows you how to create and use pointers to simple variables.

### 9-2-1. Declaring Pointers

A pointer is a numeric variable and, like all variables, must be declared before it can be used. Pointer variable names follow the same rules as other variables and must be unique. This chapter uses the convention that a pointer to the variable name is called p_name. This isn't necessary, however; you can name pointers anything you want (within C's naming rules).

A pointer declaration takes the following form:

```
typename *ptrname;
```

*typename* is any of C's variable types and indicates the type of the variable that the pointer points to. The asterisk (*) is the indirection operator, and it indicates that *ptrname* is a pointer to type *typename* and not a variable of type *typename*. Pointers can be declared along with nonpointer variables. Here are some more examples:

```
char *ch1, *ch2;          /* ch1 and ch2 both are pointers to type char
*/
float *value, percent;    /* value is a pointer to type float, and
                          /* percent is an ordinary float variable */
```

---

**NOTE:** The * symbol is used as both the indirection operator and the multiplication operator. Don't worry about the compiler's becoming confused. The context in which * is used always provides enough information so that the compiler can figure out whether you mean indirection or multiplication.

---

### 9-2-2. Initializing Pointers

Now that you've declared a pointer, what can you do with it? You can't do anything with it until you make it point to something. Like regular variables, uninitialized pointers can be used, but the results are unpredictable and potentially disastrous. Until a pointer holds the address of a variable, it isn't useful. The address doesn't get stored in the pointer by magic; your program must put it there by using the address-of operator, the ampersand (&). When placed before the name of a variable, the address-of operator returns the address of the variable. Therefore, you initialize a pointer with a statement of the form

```
pointer = &variable;
```

Look back at the example in Figure 9.3. The program statement to initialize the variable p_rate to point at the variable rate would be

```
p_rate = &rate;     /* assign the address of rate to p_rate */
```

Before the initialization, p_rate didn't point to anything in particular. After the initialization, p_rate is a pointer to rate.

### 9-2-3. Using Pointers

Now that you know how to declare and initialize pointers, you're probably wondering how to use them. The indirection operator (*) comes into play again. When the * precedes the name of a pointer, it refers to the variable pointed to.

Let's continue with the previous example, in which the pointer p_rate has been initialized to point to the variable rate. If you write *p_rate, it refers to the variable rate. If you want to print the value of rate (which is 100 in the example), you could write

```
printf("%d", rate);
```

or this:

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

In C, these two statements are equivalent. Accessing the contents of a variable by using the variable name is called *direct access*. Accessing the contents of a variable by using a pointer to the variable is called *indirect access* or *indirection*. Figure 9.4 shows that a pointer name preceded by the indirection operator refers to the value of the pointed-to variable. The variable p_rate contains the address of the variable rate and is therefore a pointer to rate.

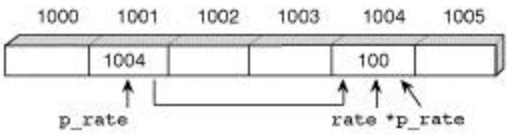


**Figure 9-4. Use of the indirection operator with pointers.**

Pause a minute and think about this material. Pointers are an integral part of the C language, and it's essential that you understand them. Pointers have confused many people, so don't worry if you're feeling a bit puzzled. If you need to review, that's fine. Maybe the following summary can help.

If you have a pointer named ptr that has been initialized to point to the variable var, the following are true:

- *ptr and var both refer to the contents of var (that is, whatever value the program has stored there).
- ptr and &var refer to the address of var.

As you can see, a pointer name without the indirection operator accesses the pointer value itself, which is, of course, the address of the variable pointed to.

Listing 9.1 demonstrates basic pointer use. You should enter, compile, and run this program.

### Listing 9.1. Basic pointer use.

```
1:  /* Demonstrates basic pointer use. */
2:
3:  #include <stdio.h>
4:
5:  /* Declare and initialize an int variable */
6:
7:  int var = 1;
8:
9:  /* Declare a pointer to int */
10:
11: int *ptr;
12:
13: main()
14: {
15:     /* Initialize ptr to point to var */
```

```
16:
17:     ptr = &var;
18:
19:     /* Access var directly and indirectly */
20:
21:     printf("\nDirect access, var = %d", var);
22:     printf("\nIndirect access, var = %d", *ptr);
23:
24:     /* Display the address of var two ways */
25:
26:     printf("\n\nThe address of var = %d", &var);
27:     printf("\nThe address of var = %d\n", ptr);
28:
29:     return 0;
30: }
Direct access, var = 1
Indirect access, var = 1
The address of var = 4264228
The address of var = 4264228
```

---

The address reported for var might not be 4264228 on your system.

---

**ANALYSIS:** In this listing, two variables are declared. In line 7, var is declared as an int and initialized to 1. In line 11, a pointer to a variable of type int is declared and named ptr. In line 17, the pointer ptr is assigned the address of var using the address-of operator (&). The rest of the program prints the values from these two variables to the screen. Line 21 prints the value of var, whereas line 22 prints the value stored in the location pointed to by ptr. In this program, this value is 1. Line 26 prints the address of var using the address-of operator. This is the same value printed by line 27 using the pointer variable, ptr.

This listing is good to study. It shows the relationship between a variable, its address, a pointer, and the dereferencing of a pointer.

---

**DO** understand what pointers are and how they work. The mastering of C requires mastering pointers.

**DON'T** use an uninitialized pointer. Results can be disastrous if you do.

---

## 9-3. Pointers and Variable Types

The previous discussion ignores the fact that different variable types occupy different amounts of memory. For the more common PC operating systems, an int takes two bytes, a float takes four bytes, and so on. Each individual byte of memory has its own address, so a multibyte variable actually occupies several addresses.

How, then, do pointers handle the addresses of multibyte variables? Here's how it works: The address of a variable is actually the address of the first (lowest) byte it occupies. This can be illustrated with an example that declares and initializes three variables:

```
int vint = 12252;
char vchar = 90;
float vfloat = 1200.156004;
```

These variables are stored in memory as shown in Figure 9.5. In this figure, the int variable occupies two bytes, the char variable occupies one byte, and the float variable occupies four bytes.

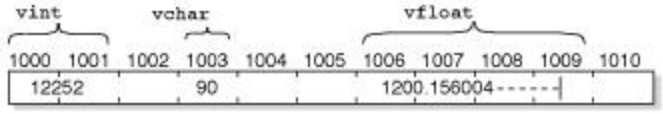


**Figure 9-5. Different types of numeric variables occupy different amounts of storage space in memory.**

Now let's declare and initialize pointers to these three variables:

```
int *p_vint;
char *p_vchar;
float *p_vfloat;
/* additional code goes here */
p_vint = &vint;
p_vchar = &vchar;
p_vfloat = &vfloat;
```

Each pointer is equal to the address of the first byte of the pointed-to variable. Thus, p_vint equals 1000, p_vchar equals 1003, and p_vfloat equals 1006. Remember, however, that each pointer was declared to point to a certain type of variable. The compiler knows that a pointer to type int points to the first of two bytes, a pointer to type float points to the first of four bytes, and so on. This is illustrated in Figure 9.6.

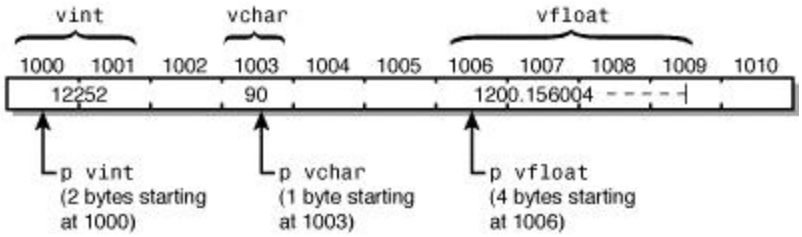


**Figure 9-6. The compiler knows the size of the vari-able that a pointer points to.**

Figures 9.5 and 9.6 show some empty memory storage locations among the three variables. This is for the sake of visual clarity. In actual practice, the C compiler stores the three variables in adjacent memory locations with no unused bytes between them.

## 9-4. Pointers and Arrays

Pointers can be useful when you're working with simple variables, but they are more helpful with arrays. There is a special relationship between pointers and arrays in C. In fact, when you use the array subscript notation that you learned on Day 8, "Using Numeric Arrays," you're really using pointers without knowing it. The following sections explain how this works.

### 9-4-1. The Array Name as a Pointer

An array name without brackets is a pointer to the array's first element. Thus, if you've declared an array data[], data is the address of the first array element.

"Wait a minute," you might be saying. "Don't you need the address-of operator to get an address?" Yes. You can also use the expression &data[0] to obtain the address of the array's first element. In C, the relationship (data == &data[0]) is true.

You've seen that the name of an array is a pointer to the array. Remember that this is a pointer constant; it can't be changed and remains fixed for the duration of program execution. This makes sense: If you changed its value, it would point elsewhere and not to the array (which remains at a fixed location in memory).

You can, however, declare a pointer variable and initialize it to point at the array. For example, the following code initializes the pointer variable p_array with the address of the first element of array[]:

```
int array[100], *p_array;
/* additional code goes here */
p_array = array;
```

Because p_array is a pointer variable, it can be modified to point elsewhere. Unlike array, p_array isn't locked into pointing at the first element of array[]. For example, it could be pointed at other elements of array[]. How would you do this? First, you need to look at how array elements are stored in memory.

## 9-4-2. Array Element Storage

As you might remember from Day 8, the elements of an array are stored in sequential memory locations with the first element in the lowest address. Subsequent array elements (those with an index greater than 0) are stored in higher addresses. How much higher depends on the array's data type (char, int, float, and so forth).

Take an array of type int. As you learned on Day 3, "Storing Data: Variables and Constants," a single int variable can occupy two bytes of memory. Each array element is therefore located two bytes above the preceding element, and the address of each array element is two higher than the address of the preceding element. A type float, on the other hand, can occupy four bytes. In an array of type float, each array element is located four bytes above the preceding element, and the address of each array element is four higher than the address of the preceding element.

Figure 9.7 illustrates the relationship between array storage and addresses for a six-element int array and a three-element float array.

By looking at Figure 9.7, you should be able to see why the following relationships are true:

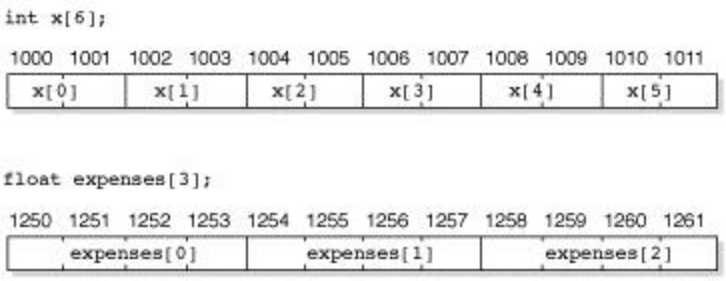


**Figure 9-7. Array storage for different array types.**

```
1: x == 1000
2: &x[0] == 1000
3: &x[1] = 1002
4: expenses == 1250
5: &expenses[0] == 1250
6: &expenses[1] == 1254
```

x without the array brackets is the address of the first element (x[0]). You can also see that x[0] is at the address of 1000. Line 2 shows this too. It can be read like this: "The address of the first element of the array x is equal to 1000." Line 3 shows that the address of the second element (subscripted as 1 in an array) is 1002. Again, Figure 9.7 can confirm this. Lines 4, 5, and 6 are virtually identical to 1, 2, and 3, respectively. They vary in the difference between the addresses of the two array elements. In the type int array x, the difference is two bytes, and in the type float array, expenses, the difference is four bytes.

How do you access these successive array elements using a pointer? You can see from these examples that a pointer must be increased by 2 to access successive elements of a type int array, and by 4 to access successive elements of a type float array. You can generalize and say that to access successive elements of an array of a particular data type, a pointer must be increased by sizeof(*datatype*). Remember from Day 3 that the sizeof() operator returns the size in bytes of a C data type.

Listing 9.2 illustrates the relationship between addresses and the elements of different type arrays by declaring arrays of type int, float, and double and by displaying the addresses of successive elements.

### Listing 9.2. Displaying the addresses of successive array elements.

```
1:  /* Demonstrates the relationship between addresses and */
2:  /* elements of arrays of different data types. */
3:
4:  #include <stdio.h>
5:
6:  /* Declare three arrays and a counter variable. */
7:
8:  int i[10], x;
9:  float f[10];
10: double d[10];
11:
12: main()
13: {
14:     /* Print the table heading */
15:
16:     printf("\t\tInteger\t\tFloat\t\tDouble");
17:
18:     printf("\n================================");
19:     printf("======================");
20:
21:     /* Print the addresses of each array element. */
22:
23:     for (x = 0; x < 10; x++)
24:         printf("\nElement %d:\t%ld\t\t%ld\t\t%ld", x, &i[x],
25:             &f[x], &d[x]);
26:
27:     printf("\n================================");
28:     printf("======================\n");
29:
30:     return 0;
31: }
               Integer     Float     Double
=============================================
Element 0:     1392        1414      1454
Element 1:     1394        1418      1462
Element 2:     1396        1422      1470
Element 3:     1398        1426      1478
Element 4:     1400        1430      1486
Element 5:     1402        1434      1494
Element 6:     1404        1438      1502
Element 7:     1406        1442      1510
Element 8:     1408        1446      1518
Element 9:     1410        1450      1526
```

**ANALYSIS:** The exact addresses that your system displays might be different from these, but the relationships are the same. In this output, there are two bytes between int elements, four bytes between float elements, and eight

bytes between double elements. (Note: Some machines use different sizes for variable types. If your machine differs, the preceding output might have different-size gaps; however, they will be consistent gaps.)

This listing takes advantage of the escape characters discussed on Day 7, "Fundamentals of Input and Output." The printf() calls in lines 16 and 24 use the tab escape character (\t) to help format the table by aligning the columns.

Looking more closely at Listing 9.2, you can see that three arrays are created in lines 8, 9, and 10. Line 8 declares array i of type int, line 9 declares array f of type float, and line 10 declares array d of type double. Line 16 prints the column headers for the table that will be displayed. Lines 18 and 19, along with lines 27 and 28, print dashed lines across the top and bottom of the table data. This is a nice touch for a report. Lines 23, 24, and 25 are a for loop that prints each of the table's rows. The number of the element x is printed first. This is followed by the address of the element in each of the three arrays.

### 9-4-3. Pointer Arithmetic

You have a pointer to the first array element; the pointer must increment by an amount equal to the size of the data type stored in the array. How do you access array elements using pointer notation? You use *pointer arithmetic*.

"Just what I don't need," you might be thinking, "another kind of arithmetic to learn!" Don't worry. Pointer arithmetic is simple, and it makes using pointers in your programs much easier. You have to be concerned with only two pointer operations: incrementing and decrementing.

### Incrementing Pointers

When you *increment* a pointer, you are increasing its value. For example, when you incre-ment a pointer by 1, pointer arithmetic automatically increases the pointer's value so that it points to the next array element. In other words, C knows the data type that the pointer points to (from the pointer declaration) and increases the address stored in the pointer by the size of the data type.

Suppose that ptr_to_int is a pointer variable to some element of an int array. If you execute the statement

```
ptr_to_int++;
```

the value of ptr_to_int is increased by the size of type int (usually 2 bytes), and ptr_to_int now points to the next array element. Likewise, if ptr_to_float points to an element of a type float array, the statement

```
ptr_to_float++;
```

increases the value of ptr_to_float by the size of type float (usually 4 bytes).

The same holds true for increments greater than 1. If you add the value *n* to a pointer, C increments the pointer by *n* array elements of the associated data type. Therefore,

```
ptr_to_int += 4;
```

increases the value stored in ptr_to_int by 8 (assuming that an integer is 2 bytes), so it points four array elements ahead. Likewise,

```
ptr_to_float += 10;
```

increases the value stored in ptr_to_float by 40 (assuming that a float is 4 bytes), so it points 10 array elements ahead.

### Decrementing Pointers

The same concepts that apply to incrementing pointers hold true for decrementing pointers. *Decrementing* a pointer is actually a special case of incrementing by adding a negative value. If you decrement a pointer with the -- or -= operators, pointer arithmetic automatically adjusts for the size of the array elements.

Listing 9.3 presents an example of how pointer arithmetic can be used to access array elements. By incrementing pointers, the program can step through all the elements of the arrays efficiently.

### Listing 9.3. Using pointer arithmetic and pointer notation to access array elements.

```
1:  /* Demonstrates using pointer arithmetic to access */
2:  /* array elements with pointer notation. */
3:
4:  #include <stdio.h>
5:  #define MAX 10
6:
7:  /* Declare and initialize an integer array. */
8:
9:  int i_array[MAX] = { 0,1,2,3,4,5,6,7,8,9 };
10:
11: /* Declare a pointer to int and an int variable. */
12:
13: int *i_ptr, count;
14:
15: /* Declare and initialize a float array. */
16:
17: float f_array[MAX] = { .0, .1, .2, .3, .4, .5, .6, .7, .8, .9 };
18:
19: /* Declare a pointer to float. */
20:
21: float *f_ptr;
22:
23: main()
24: {
25:     /* Initialize the pointers. */
26:
27:     i_ptr = i_array;
28:     f_ptr = f_array;
29:
30:     /* Print the array elements. */
31:
32:     for (count = 0; count < MAX; count++)
33:         printf("%d\t%f\n", *i_ptr++, *f_ptr++);
34:
35:     return 0;
36: }
0       0.000000
1       0.100000
2       0.200000
3       0.300000
4       0.400000
5       0.500000
6       0.600000
```

```
7       0.700000
8       0.800000
9       0.900000
```

**ANALYSIS:** In this program, a defined constant named MAX is set to 10 in line 5; it is used throughout the listing. In line 9, MAX is used to set the number of elements in an array of ints named i_array. The elements in this array are initialized at the same time that the array is declared. Line 13 declares two additional int variables. The first is a pointer named i_ptr. You know this is a pointer because an indirection operator (*) is used. The other variable is a simple type int variable named count. In line 17, a second array is defined and initialized. This array is of type float, contains MAX values, and is initialized with float values. Line 21 declares a pointer to a float named f_ptr.

The main() function is on lines 23 through 36. The program assigns the beginning address of the two arrays to the pointers of their respective types in lines 27 and 28. Remember, an array name without a subscript is the same as the address of the array's beginning. A for statement in lines 32 and 33 uses the int variable count to count from 0 to the value of MAX. For each count, line 33 dereferences the two pointers and prints their values in a printf() function call. The increment operator then increments each of the pointers so that each points to the next element in the array before continuing with the next iteration of the for loop.

You might be thinking that this program could just as well have used array subscript notation and dispensed with pointers altogether. This is true, and in simple programming tasks like this, the use of pointer notation doesn't offer any major advantages. As you start to write more complex programs, however, you should find the use of pointers advantageous.

Remember that you can't perform incrementing and decrementing operations on pointer constants. (An array name without brackets is a *pointer constant*.) Also remember that when you're manipulating pointers to array elements, the C compiler doesn't keep track of the start and finish of the array. If you're not careful, you can increment or decrement the pointer so that it points somewhere in memory before or after the array. Something is stored there, but it isn't an array element. You should keep track of pointers and where they're pointing.

## Other Pointer Manipulations

The only other pointer arithmetic operation is called *differencing,* which refers to subtracting two pointers. If you have two pointers to different elements of the same array, you can subtract them and find out how far apart they are. Again, pointer arithmetic automatically scales the answer so that it refers to array elements. Thus, if ptr1 and ptr2 point to elements of an array (of any type), the following expression tells you how far apart the elements are:

```
ptr1 - ptr2
```

Pointer comparisons are valid only between pointers that point to the same array. Under these circumstances, the relational operators ==, !=, >, <, >=, and <= work properly. Lower array elements (that is, those having a lower subscript) always have a lower address than higher array elements. Thus, if ptr1 and ptr2 point to elements of the same array, the comparison

```
ptr1 < ptr2
```

is true if ptr1 points to an earlier member of the array than ptr2 does.

This covers all allowed pointer operations. Many arithmetic operations that can be performed with regular variables, such as multiplication and division, don't make sense with pointers. The C compiler doesn't allow them. For example, if ptr is a pointer, the statement

```
ptr *= 2;
```

generates an error message. As Table 9.1 indicates, you can do a total of six operations with a pointer, all of which have been covered in this chapter.

**Table 9.1. Pointer operations.**

| Operation | Description |
|---|---|
| Assignment | You can assign a value to a pointer. The value should be an address, obtained with the address-of operator (&) or from a pointer constant (array name). |
| Indirection | The indirection operator (*) gives the value stored in the pointed-to location. |
| Address of | You can use the address-of operator to find the address of a pointer, so you can have pointers to pointers. This is an advanced topic and is covered on Day 15, "Pointers: Beyond the Basics." |
| Incrementing | You can add an integer to a pointer in order to point to a different memory location. |
| Decrementing | You can subtract an integer from a pointer in order to point to a different memory location. |
| Differencing | You can subtract an integer from a pointer in order to point to a different memory location. |
| Comparison | Valid only with two pointers that point to the same array. |

## 9-5. Pointer Cautions

When you're writing a program that uses pointers, you must avoid one serious error: using an uninitialized pointer on the left side of an assignment statement. For example, the following statement declares a pointer to type int:

```
int *ptr;
```

This pointer isn't yet initialized, so it doesn't point to anything. To be more exact, it doesn't point to anything *known*. An uninitialized pointer has some value; you just don't know what it is. In many cases, it is zero. If you use an uninitialized pointer in an assignment statement, this is what happens:

```
*ptr = 12;
```

The value 12 is assigned to whatever address ptr points to. That address can be almost anywhere in memory--where the operating system is stored or somewhere in the program's code. The 12 that is stored there might overwrite some important information, and the result can be anything from strange program errors to a full system crash.

The left side of an assignment statement is the most dangerous place to use an uninitialized pointer. Other errors, although less serious, can also result from using an uninitialized pointer anywhere in your program, so be sure your program's pointers are properly initialized before you use them. You must do this yourself. The compiler won't do this for you!

---

**DON'T** try to perform mathematical operations such as division, multiplication, and modulus on pointers. Adding (incrementing) and subtracting (differencing) pointers are acceptable.

**DON'T** forget that subtracting from or adding to a pointer changes the pointer based on the size of the data type it points to. It doesn't change it by 1 or by the number being added (unless it's a pointer to a one-byte character).

**DO** understand the size of variable types on your computer. As you can begin to see, you need to know variable sizes when working with pointers and memory.

**DON'T** try to increment or decrement an array variable. Assign a pointer to the beginning address of the array and increment it (see Listing 9.3).

---

## 9-6. Array Subscript Notation and Pointers

An array name without brackets is a pointer to the array's first element. Therefore, you can access the first array element using the indirection operator. If array[] is a declared array, the expression *array is the array's first element, *(array + 1) is the array's second element, and so on. If you generalize for the entire array, the following relationships hold true:

```
*(array) == array[0]
*(array + 1) == array[1]
*(array + 2) == array[2]
...
*(array + n) == array[n]
```

This illustrates the equivalence of array subscript notation and array pointer notation. You can use either in your programs; the C compiler sees them as two different ways of accessing array data using pointers.

## 9-7. Passing Arrays to Functions

This chapter has already discussed the special relationship that exists in C between pointers and arrays. This relationship comes into play when you need to pass an array as an argument to a function. The only way you can pass an array to a function is by means of a pointer.

As you learned on Day 5, "Functions: The Basics," an argument is a value that the calling program passes to a function. It can be an int, a float, or any other simple data type, but it must be a single numerical value. It can be a single array element, but it can't be an entire array. What if you need to pass an entire array to a function? Well, you can have a pointer to an array, and that pointer is a single numeric value (the address of the array's first element). If you pass that value to a function, the function knows the address of the array and can access the array elements using pointer notation.

Consider another problem. If you write a function that takes an array as an argument, you want a function that can handle arrays of different sizes. For example, you could write a function that finds the largest element in an integer array. The function wouldn't be much use if it were limited to dealing with arrays of one fixed size.

How does the function know the size of the array whose address it was passed? Remember, the value passed to a function is a pointer to the first array element. It could be the first of 10 elements or the first of 10,000. There are two methods of letting a function know an array's size.

You can identify the last array element by storing a special value there. As the function processes the array, it looks for that value in each element. When the value is found, the end of the array has been reached. The disadvantage of this method is that it forces you to reserve a value as the end-of-array indicator, reducing the flexibility you have for storing real data in the array.

The other method is more flexible and straightforward, and it's the one used in this book: Pass the function the array size as an argument. This can be a simple type int argument. Thus, the function is passed two arguments: a pointer to the first array element, and an integer specifying the number of elements in the array.

Listing 9.4 accepts a list of values from the user and stores them in an array. It then calls a function named largest(), passing the array (both pointer and size). The function finds the largest value in the array and returns it to the calling program.

### Listing 9.4. Passing an array to a function.

```
1:  /* Passing an array to a function. */
2:
3:  #include <stdio.h>
4:
5:  #define MAX 10
6:
7:  int array[MAX], count;
8:
9:  int largest(int x[], int y);
10:
11: main()
12: {
13:     /* Input MAX values from the keyboard. */
14:
15:     for (count = 0; count < MAX; count++)
16:     {
17:         printf("Enter an integer value: ");
18:         scanf("%d", &array[count]);
19:     }
20:
21:     /* Call the function and display the return value. */
22:     printf("\n\nLargest value = %d\n", largest(array, MAX));
23:
24:     return 0;
25: }
26: /* Function largest() returns the largest value */
27: /* in an integer array */
28:
29: int largest(int x[], int y)
30: {
31:     int count, biggest = -12000;
32:
33:     for ( count = 0; count < y; count++)
34:     {
35:         if (x[count] > biggest)
36:             biggest = x[count];
37:     }
38:
39:     return biggest;
40: }
Enter an integer value: 1
Enter an integer value: 2
Enter an integer value: 3
Enter an integer value: 4
Enter an integer value: 5
Enter an integer value: 10
```

```
Enter an integer value: 9
Enter an integer value: 8
Enter an integer value: 7
Enter an integer value: 6
Largest value = 10
```

**ANNALYSIS:** A function prototype in line 9 and a function header in line 29 are nearly identical except for a semicolon:

```
int largest(int x[], int y)
```

Most of this line should make sense to you: largest() is a function that returns an int to the calling program; its second argument is an int represented by the parameter y. The only thing new is the first parameter, int x[], which indicates that the first argument is a pointer to type int, represented by the parameter x. You also could write the function declaration and header as follows:

```
int largest(int *x, int y);
```

This is equivalent to the first form; both int x[] and int *x mean "pointer to int." The first form might be preferable, because it reminds you that the parameter represents a pointer to an array. Of course, the pointer doesn't know that it points to an array, but the function uses it that way.

Now look at the function largest(). When it is called, the parameter x holds the value of the first argument and is therefore a pointer to the first element of the array. You can use x anywhere an array pointer can be used. In largest(), the array elements are accessed using subscript notation in lines 35 and 36. You also could use pointer notation, rewriting the if loop like this:

```
for (count = 0; count < y; count++)
{
   if (*(x+count) > biggest)
      biggest = *(x+count);
}
```

Listing 9.5 shows the other way of passing arrays to functions.

### Listing 9.5. An alternative way of passing an array to a function.

```
1:  /* Passing an array to a function. Alternative way. */
2:
3:  #include <stdio.h>
4:
5:  #define MAX 10
6:
7:  int array[MAX+1], count;
8:
9:  int largest(int x[]);
10:
11: main()
12: {
13:     /* Input MAX values from the keyboard. */
14:
15:     for (count = 0; count < MAX; count++)
16:     {
```

```
17:         printf("Enter an integer value: ");
18:         scanf("%d", &array[count]);
19:
20:         if ( array[count] == 0 )
21:             count = MAX;              /* will exit for loop */
22:     }
23:     array[MAX] = 0;
24:
25:     /* Call the function and display the return value. */
26:     printf("\n\nLargest value = %d\n", largest(array));
27:
28:     return 0;
29: }
30: /* Function largest() returns the largest value */
31: /* in an integer array */
32:
33: int largest(int x[])
34: {
35:     int count, biggest = -12000;
36:
37:     for ( count = 0; x[count] != 0; count++)
38:     {
39:         if (x[count] > biggest)
40:             biggest = x[count];
41:     }
42:
43:     return biggest;
44: }
Enter an integer value: 1
Enter an integer value: 2
Enter an integer value: 3
Enter an integer value: 4
Enter an integer value: 5
Enter an integer value: 10
Enter an integer value: 9
Enter an integer value: 8
Enter an integer value: 7
Enter an integer value: 6
Largest value = 10
```

Here is the output from running the program a second time:

```
Enter an integer value: 10
Enter an integer value: 20
Enter an integer value: 55
Enter an integer value: 3
Enter an integer value: 12
Enter an integer value: 0
Largest value = 55
```

This program uses a largest() function that has the same functionality as Listing 9.4. The difference is that only the array tag is needed. The for loop in line 37 continues looking for the largest value until it encounters a 0, at which point it knows it is done.

Looking at the early parts of this program, you can see the differences between Listing 9.4 and Listing 9.5. First, in line 7 you need to add an extra element to the array to store the value that indicates the end. In lines 20 and 21, an if statement is added to see whether the user entered 0, thus signaling that he is done entering values. If 0 is entered, count is set to its maximum value so that the for loop can be exited cleanly. Line 23 ensures that the last element is 0 in case the user entered the maximum number of values (MAX).

By adding the extra commands when entering the data, you can make the largest() function work with any size of array; however, there is one catch. What happens if you forget to put a 0 at the end of the array? largest() continues past the end of the array, comparing values in memory until it finds a 0.

As you can see, passing an array to a function is not particularly difficult. You simply pass a pointer to the array's first element. In most situations, you also need to pass the number of elements in the array. In the function, the pointer value can be used to access the array elements with either subscript or pointer notation.

---

**WARNING:** Recall from Day 5 that when a simple variable is passed to a function, only a copy of the variable's value is passed. The function can use the value but can't change the original variable because it doesn't have access to the variable itself. When you pass an array to a function, things are different. A function is passed the array's address, not just a copy of the values in the array. The code in the function works with the actual array elements and can modify the values stored in the array.

---

## 9-8. Summary

This chapter introduced you to pointers, a central part of C programming. A pointer is a variable that holds the address of another variable; a pointer is said to "point to" the variable whose address it holds. The two operators needed with pointers are the address-of operator (&) and the indirection operator (*). When placed before a variable name, the address-of operator returns the variable's address. When placed before a pointer name, the indirection operator returns the contents of the pointed-to variable.

Pointers and arrays have a special relationship. An array name without brackets is a pointer to the array's first element. The special features of pointer arithmetic make it easy to access array elements using pointers. Array subscript notation is in fact a special form of pointer notation.

You also learned to pass arrays as arguments to functions by passing a pointer to the array. Once the function knows the array's address and length, it can access the array elements using either pointer notation or subscript notation.

## Q&A

**Q Why are pointers so important in C?**
**A** Pointers give you more control over the computer and your data. When used with functions, pointers let you change the values of variables that were passed, regardless of where they originated. On Day 15, you will learn additional uses for pointers.
**Q How does the compiler know the difference between * for multiplication, for dereferencing, and for declaring a pointer?**
**A** The compiler interprets the different uses of the asterisk based on the context in which it is used. If the statement being evaluated starts with a variable type, it can be assumed that the asterisk is for declaring a

pointer. If the asterisk is used with a variable that has been declared as a pointer, but not in a variable declaration, the asterisk is assumed to dereference. If it is used in a mathematical expression, but not with a pointer variable, the asterisk can be assumed to be the multiplication operator.

**Q What happens if I use the address-of operator on a pointer?**

**A** You get the address of the pointer variable. Remember, a pointer is just another variable that holds the address of the variable to which it points.

**Q Are variables always stored in the same location?**

**A** No. Each time a program runs, its variables can be stored at different addresses within the computer. You should never assign a constant address value to a pointer.

## Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned.

## Quiz

**1.** What operator is used to determine the address of a variable?
**2.** What operator is used to determine the value at the location pointed to by a pointer?
**3.** What is a pointer?
**4.** What is indirection?
**5.** How are the elements of an array stored in memory?
**6.** Show two ways to obtain the address of the first element of the array data[].
**7.** If an array is passed to a function, what are two ways to know where the end of that array is?
**8.** What are the six operations covered in this chapter that can be accomplished with a pointer?
**9.** Assume that you have two pointers. If the first points to the third element in an array of ints and the second points to the fourth element, what value is obtained if you subtract the first pointer from the second? (Assume that the size of an integer is 2 bytes.)
**10.** Assume that the array in question 9 is of float values. What value is obtained if the two pointers are subtracted? (Assume that the size of a float is 2 bytes.)

## Exercises

**1.** Show a declaration for a pointer to a type char variable. Name the pointer char_ptr.
**2.** If you have a type int variable named cost, how would you declare and initialize a pointer named p_cost that points to that variable?
**3.** Continuing with exercise 2, how would you assign the value 100 to the variable cost using both direct access and indirect access?
**4.** Continuing with exercise 3, how would you print the value of the pointer, plus the value being pointed to?
**5.** Show how to assign the address of a float value called radius to a pointer.
**6.** Show two ways to assign the value 100 to the third element of data[].
**7.** Write a function named sumarrays() that accepts two arrays as arguments, totals all values in both arrays, and returns the total to the calling program.
**8.** Use the function created in exercise 7 in a simple program.
**9.** Write a function named addarrays() that accepts two arrays that are the same size. The function should add each element in the arrays together and place the values in a third array.
**10. ON YOUR OWN:** Modify the function in exercise 9 to return a pointer to the array containing the totals. Place this function in a program that also displays the values in all three arrays.