Pointers

C++ reveals its true power through pointer variables. Pointer variables (or *pointers*, as they generally are called) are variables that contain addresses of other variables. All variables you have seen so far have held data values. You understand that variables hold various data types: character, integer, floating-point, and so on. Pointer variables contain the location of regular data variables; they in effect point to the data because they hold the address of the data.

When first learning C++, students of the language tend to shy away from pointers, thinking that pointers will be difficult. Pointers do not have to be difficult. In fact, after you work with them for a while, you will find they are easier to use than arrays (and much more flexible).

This chapter introduces the following concepts:

- Pointers
- Pointers of different data types
- The "address of" (&) operator
- The dereferencing (*) operator
- Arrays of pointers

Pointers offer a highly efficient means of accessing and changing data. Because pointers contain the actual address of your data, your compiler has less work to do when finding that data in memory. Pointers do not have to link data to specific variable names. A pointer can point to an unnamed data value. With pointers, you gain a "different view" of your data.

Introduction to Pointer Variables

Pointers contain addresses of other variables.

Pointers are variables. They follow all the normal naming rules of regular, nonpointer variables. As with regular variables, you must declare pointer variables before using them. There is a type of pointer for every data type in C++; there are integer pointers, character pointers, floating-point pointers, and so on. You can declare global pointers or local pointers, depending on where you declare them.

About the only difference between pointer variables and regular variables is the data they hold. Pointers do not contain data in the usual sense of the word. Pointers contain addresses of data. If you need a quick review of addresses and memory, see Appendix A, "Memory Addressing, Binary, and Hexadecimal Review."

There are two pointer operators in C++:

- `&` The "address of" operator
- `*` The dereferencing operator

Don't let these operators throw you; you might have seen them before! The `&` is the bitwise AND operator (from Chapter 11, "Additional C++ Operators") and the `*` means, of course, multiplication. These are called *overloaded* operators. They perform more than one function, depending on how you use them in your programs. C++ does not confuse `*` for multiplication when you use it as a dereferencing operator with pointers.

Any time you see the & used with pointers, think of the words "address of." The & operator always produces the memory address of whatever it precedes. The * operator, when used with pointers, either declares a pointer or dereferences the pointer's value. The next section explains each of these operators.

Declaring Pointers

Because you must declare all pointers before using them, the best way to begin learning about pointers is to understand how to declare and define them. Actually, declaring pointers is almost as easy as declaring regular variables. After all, pointers are variables.

If you must declare a variable that holds your age, you could do so with the following variable declaration:

```
int age=30;        // Declare a variable to hold my age.
```

Declaring age like this does several things. It enables C++ to identify a variable called age, and to reserve storage for that variable. Using this format also enables C++ to recognize that you will store only integers in age, not floating-point or double floating-point data. The declaration also requests that C++ store the value of 30 in age after it reserves storage for age.

Where did C++ store age in memory? As the programmer, you should not really care where C++ stores age. You do not have to know the variable's address because you will never refer to age by its address. If you want to calculate with or print age, you call it by its name, age.

TIP: Make your pointer variable names meaningful. The name file_ptr makes more sense than x13 for a file-pointing variable, although either name is allowed.

Suppose you want to declare a pointer variable. This pointer variable will not hold your age, but it will point to age, the variable that holds your age. (Why you would want to do this is explained in this and the next few chapters.) p_age might be a good name for the pointer variable. Figure 26.1 illustrates what you want to do. The

figure assumes C++ stored age at the address 350,606. Your C++ compiler, however, arbitrarily determines the address of age, so it could be anything.

Figure 26.1. p_age contains the address of age; p_age points to the age variable.

The name p_age has nothing to do with pointers, except that it is the name you made up for the pointer to age. Just as you can name variables anything (as long as the name follows the legal naming rules of variables), p_age could just as easily have been named house, x43344, space_trek, or whatever else you wanted to call it. This reinforces the idea that a pointer is just a variable you reserve in your program. Create meaningful variable names, even for pointer variables. p_age is a good name for a variable that points to age (as would be ptr_age and ptr_to_age).

To declare the p_age pointer variable, you must program the following:

```
int * p_age;              // Declares an integer pointer.
```

Similar to the declaration for age, this declaration reserves a variable called p_age. The p_age variable is not a normal integer variable, however. Because of the dereferencing operator, *, C++ knows this is to be a pointer variable. Some C++ programmers prefer to declare such a variable without a space after the *, as follows:

```
int *p_age;               // Declares an integer pointer.
```

Either method is okay, but you must remember the * is *not* part of the name. When you later use p_age, you will not prefix the name with the *, unless you are dereferencing it at the time (as later examples show).

TIP: Whenever the dereferencing operator, *, appears in a variable definition, the variable being declared is *always* a pointer variable.

Consider the declaration for p_age if the asterisk were not there: C++ would think you were declaring a regular integer variable. The * is important, because it tells C++ to interpret p_age as a pointer variable, not as a normal, data variable.

Assigning Values to Pointers

Pointers can point only to data of their own type.

p_age is an integer pointer. This is very important. p_age can point only to integer values, never to floating-point, double floating-point, or even character variables. If you needed to point to a floating-point variable, you might do so with a pointer declared as

```
float *point;      // Declares a floating-point pointer.
```

As with any automatic variable, C++ does not initialize pointers when you declare them. If you declared p_age as previously described, and you wanted p_age to point to age, you would have to explicitly assign p_age to the address of age. The following statement does this:

```
p_age = &age;     // Assign the address of age to p_age.
```

What value is now in p_age? You do not know exactly, but you know it is the address of age, wherever that is. Rather than assign the address of age to p_age with an assignment operator, you can declare and initialize pointers at the same time. These lines declare and initialize both age and p_age:

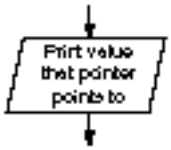
```
int age=30;       // Declares a regular integer
                  // variable, putting 30 in it.
```

```
int *p_age=&age;    // Declares an integer pointer,
                    // initializing it with the address
                    // of p_age.
```

These two lines produce the variables described in Figure 26.1.

If you wanted to print the value of age, you could do so with the following cout:

```
cout << age;            // Prints the value of age.
```



You also can print the value of age like this:

```
cout << *p_age;         // Dereferences p_age.
```

The dereference operator produces a value that tells the pointer where to point. Without the *, the last cout would print an address (the address of age). With the *, the cout prints the value at that address.

You can assign a different value to age with the following statement:

```
age=41;                 // Assigns a new value to age.
```

You also can assign a value to age like this:

```
*p_age=41;
```

This declaration assigns 41 to the value to which p_age points.

TIP: The * appears before a pointer variable in only two places—when you declare a pointer variable, and when you dereference a pointer variable (to find the data it points to).

Pointers and Parameters

Now that you understand the pointer's * and & operators, you can finally see why scanf()'s requirements were not as strict as they first seemed. While passing a regular variable to scanf(), you had to prefix the variable with the & operator. For instance, the following scanf() gets three integer values from the user:

```
scanf(" %d %d %d", &num1, &num2, &num3);
```

This `scanf()` does not pass the three variables, but passes the addresses of the three variables. Because `scanf()` knows the exact locations of these parameters in memory (because their addresses were passed), it goes to those addresses and puts the keyboard input values into those addresses.

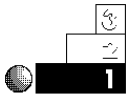
This is the only way `scanf()` could work. If you passed these variables by copy, without putting the "address of" operator (`&`) before them, `scanf()` would get the keyboard input and fill a *copy* of the variables, but not the actual variables `num1`, `num2`, and `num3`. When `scanf()` then returned control to your program, you would not have the input values. Of course, the `cin` operator does not have the ampersand (`&`) requirement and is easier to use for most C++ programs.

You might recall from Chapter 18, "Passing Values," that you can override C++'s normal default of passing by copy (or "by value"). To pass by address, receive the variable preceded by an `&` in the receiving function. The following function receives `tries` by address:

```
pr_it(int &tries);    // Receive integer tries in pr_it() by
                      // address (pr_it would normally receive
                      // tries by copy).
```

Now that you understand the `&` and `*` operators, you can understand completely the passing of nonarray parameters by address to functions. (Arrays default to passing by address without requiring that you use `&`.)

Examples



1. The following section of code declares three regular variables of three different data types, and three corresponding pointer variables:

```
char initial= 'Q';     // Declares three regular variables
int num=40;            // of three different types.
float sales=2321.59;
```

```
char *p_initial=&initial;   // Declares three pointers.
int * ptr_num=&num;         // Pointer names and spacing
float * sales_add = &sales; // after * are not critical.
```

2. Just like regular variables, you can initialize pointers with assignment statements. You do not have to initialize them when you declare them. The next few lines of code are equivalent to the code in Example 1:

```
char initial;       // Declares three regular variables
int num;            // of three different types.
float sales;

char *p_initial;    // Declares three pointers but does
int * ptr_num;      // not initialize them yet.
float * sales_add;

initial='Q';        // Initializes the regular variables
num=40;             // with values.
sales=2321.59;

p_initial=&initial;     // Initializes the pointers with
ptr_num=&num;           // the addresses of their
sales_add=&sales;       // corresponding variables.
```

Notice that you do not put the * operator before the pointer variable names when assigning them values. You would prefix a pointer variable with the * only if you were dereferencing it.

NOTE: In this example, the pointer variables could have been assigned the addresses of the regular variables before the regular variables were assigned values. There would be no difference in the operation. The pointers are assigned the addresses of the regular variables no matter what the data in the regular variables are.

Keep the data type of each pointer consistent with its corresponding variable. Do not assign a floating-point variable to an integer's address. For instance, you cannot make the following assignment statement:

```
p_initial = &sales;        // Invalid pointer assignment.
```

because `p_initial` can point only to character data, not to floating-point data.

3. The following program is an example you should study closely. It shows more about pointers and the pointer operators, `&` and `*`, than several pages of text can do.

```
// Filename: C26POINT.CPP
// Demonstrates the use of pointer declarations
// and operators.
#include <iostream.h>

void main()
{
   int num=123;            // A regular integer variable.
   int *p_num;             // Declares an integer pointer.

   cout << "num is " << num << "\n";  // Prints value of num.
   cout << "The address of num is " << &num << "\n";
                           // Prints num's location.
   p_num = &num;           // Puts address of num in p_num,
                           // in effect making p_num point
                           // to num.
                           // No * in front of p_num.
   cout << "*p_num is " << *p_num << "\n"; // Prints value
                                          // of num.
   cout << "p_num is " << p_num << "\n";   // Prints location
                                          // of num.

   return;
}
```

Here is the output from this program:

```
num is 123
The address of num is 0x8fbd0ffe
*p_num is 123
p_num is 0x8fbd0ffe
```

If you run this program, you probably will get different results for the value of `p_num` because your compiler will place `num` at a different location, depending on your memory setup. The value of `p_num` prints in hexadecimal because it is an address of memory. The actual address does not matter, however. Because the pointer `p_num` always contains the address of `num`, and because you can dereference `p_num` to get `num`'s value, the actual address is not critical.

4. The following program includes a function that swaps the values of any two integers passed to it. You might recall that a function can return only a single value. Therefore, before now, you could not write a function that changed two different values and returned both values to the calling function.

   To swap two variables (reversing their values for sorting, as you saw in Chapter 24, "Array Processing"), you need the ability to pass both variables by address. Then, when the function reverses the variables, the calling function's variables also are swapped.

   Notice the function's use of dereferencing operators before each occurrence of `num1` and `num2`. It does not matter at which address `num1` and `num2` are stored, but you must make sure that you dereference whatever addresses were passed to the function.

   Be sure to receive arguments with the prefix `&` in functions that receive by address, as done here.

Identify the program and include the I/O header file. This program swaps two integers, so initialize two integer variables in `main()`. *Pass the variables to the swapping function, called* `swap_them`, *then switch their values. Print the results of the swap in* `main()`.


```
// Filename: C26SWAP.CPP
// Program that includes a function that swaps
// any two integers passed to it
#include <iostream.h>
void swap_them(int &num1, int &num2);

void main()
{
     int i=10, j=20;
     cout << "\n\nBefore swap, i is " << i <<
             " and j is " << j << "\n\n";
     swap_them(i, j);
     cout << "\n\nAfter swap, i is " << i <<
              " and j is " << j << "\n\n";
     return;
}
void swap_them(int &num1, int &num2)
{
   int temp;           // Variable that holds
                       // in-between swapped value.
   temp = num1;        // The calling function's variables
   num1 = num2;        // (and not copies of them) are
   num2 = temp;        // changed in this function.
   return;
}
```

Arrays of Pointers

If you have to reserve many pointers for many different values, you might want to declare an array of pointers. You know that you can reserve an array of characters, integers, long integers, and floating-point values, as well as an array of every other data type available. You also can reserve an array of pointers, with each pointer being a pointer to a specific data type.

The following reserves an array of 10 integer pointer variables:

```
int *iptr[10]; // Reserves an array of 10 integer pointers
```

Figure 26.2 shows how C++ views this array. Each element holds an address (after being assigned values) that points to other values in memory. Each value pointed to must be an integer. You can assign an element from `iptr` an address just as you would for nonarray pointer variables. You can make `iptr[4]` point to the address of an integer variable named `age` by assigning it like this:

```
iptr[4] = &age;     // Make iptr[4] point to address of age.
```

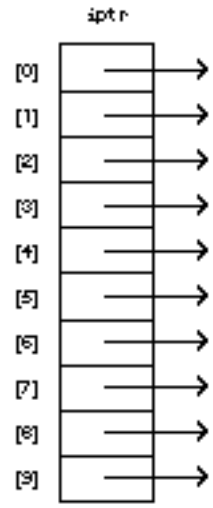


Figure 26.2. An array of 10 integer pointers.

The following reserves an array of 20 character pointer variables:

```
char *cpoint[20];          // Array of 20 character pointers.
```

Again, the asterisk is not part of the array name. The asterisk lets C++ know that this is an array of integer pointers and not just an array of integers.

Some beginning C++ students get confused when they see such a declaration. Pointers are one thing, but reserving storage for arrays of pointers tends to bog novices down. However, reserving storage for arrays of pointers is easy to understand. Remove the asterisk from the previous declaration as follows,

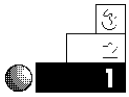
```
char cpoint[20];
```

and what do you have? You have just reserved a simple array of 20 characters. Adding the asterisk tells C++ to go one step further: rather than an array of character variables, you want an array of character pointing variables. Rather than having each element be a character variable, you have each element hold an address that points to characters.

Reserving arrays of pointers will be much more meaningful after you learn about structures in the next few chapters. As with regular, nonpointing variables, an array makes processing several pointer variables much easier. You can use a subscript to reference each variable (element) without having to use a different variable name for each value.

Review Questions

Answers to review questions are in Appendix B.



1. What type of variable is reserved in each of the following?

   a. `int *a;`

   b. `char * cp;`

   c. `float * dp;`

2. What words should come to mind when you see the `&` operator?

3. What is the dereferencing operator?

4. How would you assign the address of the floating-point variable `salary` to a pointer called `pt_sal`?

5. True or false: You must define a pointer with an initial value when declaring it.

6. In both of the following sections of code:

```
int i;
int * pti;
i=56;
pti = &i;
```

and

```
int i;
int * pti;
pti = &i;          // These two lines are reversed
i=56;              // from the preceding example.
```

is the value of `pti` the same after the fourth line of each section?

7. In the following section of code:

```
float pay;
float *ptr_pay;
pay=2313.54;
ptr_pay = &pay;
```

What is the value of each of the following (answer "invalid" if it cannot be determined):

a. `pay`

b. `*ptr_pay`

c. `*pay`

d. `&pay`

8. What does the following declare?

```
double *ara[4][6];
```

a. An array of double floating-point values

b. An array of double floating-point pointer variables

c. An invalid declaration statement

NOTE: Because this is a theory-oriented chapter, review exercises are saved until you master Chapter 27, "Pointers and Arrays."

Summary

Declaring and using pointers might seem troublesome at this point. Why assign `*p_num` a value when it is easier (and clearer) to assign a value directly to `num`? If you are asking yourself that question, you probably understand everything you should from this chapter and are ready to begin learning the true power of pointers: combining pointers and array processing.