10 Pointers, enum, and Structures

OBJECTIVES

- *Understand the basics of pointers.*
- *Declare pointers.*
- *Use the address-of and dereferencing operators.*
- *Use pointers with character arrays.*
- *Use subscript notation.*
- *Use enum.*
- *Understand what structures are and how to use them.*

Overview

A pointer is a variable or constant that holds a memory address. Pointers may be a new term to you, or you may have heard that they are difficult to understand. As with any new concept, however, once you become familiar with the principles and how to apply them you will see that pointers are not difficult. This chapter will cover the basics of pointers and give you a firm foundation that you will use in chapters to come.

You will also take another look at character arrays. You will learn how to access individual characters using a method called subscript notation and using pointers. Finally, you will use a feature of C++ that lets you create your own data types.

CHAPTER 10, SECTION 1

Pointer Basics

C++'s extensive support of pointers is one of the things that makes it such a powerful language. In this chapter, you will not see all of the power that pointers bring to the language. You will, however, learn the basics of pointers that you need to unleash that power in later chapters.

REFRESH YOUR MEMORY ABOUT MEMORY

Each byte of a microcomputer's memory has a unique address. The address is just a number. Memory addresses start at zero and count up from there. Actually, the way memory is organized and the way addresses are assigned varies among computers. The important thing to know is that each byte is numbered in order. For example, memory location 221345 is next to memory location 221346.

Programming would be more difficult if you had to remember the addresses where you stored your data. Instead, the compiler lets you assign names to memory locations when you declare variables.

WHAT IS A POINTER?

A *pointer* is a variable or constant that holds a memory address. In fact, you have been using pointers already. For example, when you declare a character array like the one shown below, `state_code` is a pointer to the first character in the character array.

```
char state_code[3];
```

Figure 10-1 shows how the character array looks in memory.

The pointer occupies four bytes of RAM. The pointer stores the value 140003, which is the memory location where the character array begins.

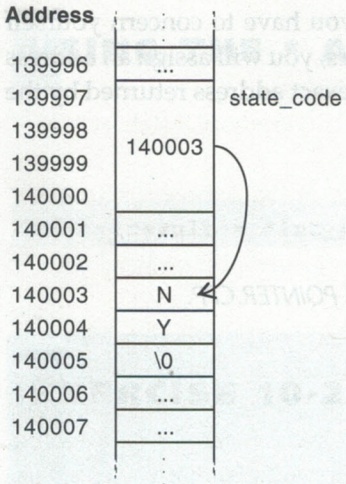

FIGURE 10-1
Declaring a character array creates a pointer to the first character contained in the array.

Note

Pointer size varies among computers and operating systems. For our purposes, the size of pointers is unimportant.

Pointers can point to more than arrays. You can create a pointer that points to any data type. For example, suppose you want a pointer that points to an integer. Figure 10-2 shows how the actual integer is stored, and how the pointer variable points to the integer.

In this example, the integer **i** (with value 3) is stored in two bytes of memory at address 216801. The pointer **iptr** points to the variable **i**.

You may be wondering why anyone would want a pointer to an integer. At this level, pointers may seem "pointless." In later chapters, however, you will learn how to put the power of pointers to work for you when more advanced methods of handling data are discussed.

DECLARING POINTERS

The code below shows how an integer and pointer like the one in Figure 10-2 is declared.

```
int main()
{
  int i;        // declare an integer i
  int *iptr;    // declare a pointer to an integer

  iptr = &i;          // initialize the pointer to the address of i
  i = 3;              // initialize i to 3;
  return 0;
}
```

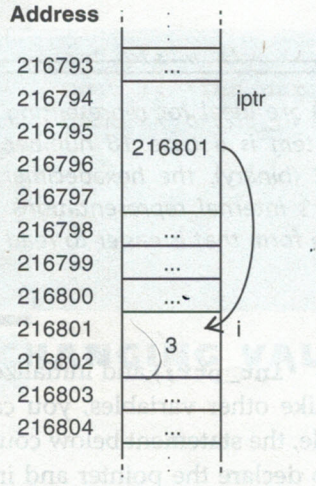

FIGURE 10-2
A pointer can point to an integer or other data type.

Working with pointers requires the use of two new operators: the ***dereferencing operator*** (*) and the ***address-of operator*** (&). Let's examine what is accomplished by the code above.

The statement **int *iptr;** declares a pointer by preceding the variable name with a dereferencing operator (*). Notice this is the same symbol used to indicate multiplication; however, your compiler can tell the difference by the way it is used.

Notice that pointers have types just like other variables. The pointer type must match the type of data you intend to point to. In the example above, an int pointer is declared to point to an int variable. The * before the variable name tells the compiler that we want to declare a pointer, rather than a regular variable. The variable **iptr** cannot hold just any value. It must hold the memory address of an integer. If you try to make a pointer point to a variable of a type other than the type of the pointer, you will get an error when you try to compile the program.

Like any other variable, a pointer begins with a random value and must be initialized to "point" to something. The statement **iptr = &i;** is what makes **iptr** point to **i**. Reading the statement as "**iptr** is assigned the address of **i**" helps the statement make sense. The address-of operator (&) returns the "address of" the variable rather than the variable's contents.

Just because you use pointers does not mean you have to concern yourself with exact memory locations. When you use pointers, you will assign an address to a pointer using the address-of operator (&). The exact address returned by the operator is of importance only to the computer.

EXERCISE 10-1 DECLARING POINTERS

1. Enter the program below. Save the source code as *POINTER.CPP*.

```
#include <iostream.h> // necessary for cout command

int main()
{
  int i, j, k;     // declare three integers
  int *int_ptr;    // declare a pointer to an integer

  i = 1;
  j = 2;
  k = 3;
  int_ptr = &i;     // initialize the pointer to point to i

  cout << "i = " << i << '\n';
  cout << "j = " << j << '\n';
  cout << "k = " << k << '\n';
  cout << "int_ptr = " << int_ptr << '\n';
  return 0;
}
```

2. Compile and run the program to see that **i, j,** and **k** hold the values you expect. The variable **int_ptr** outputs a memory address to your screen when you print it. The address will probably print in a form called hexadecimal, which is a combination of numbers and letters. Learning the exact meaning of the address is of little importance: just realize that it is the address where the variable **i** is stored.
3. Leave the source code file on the screen for the next exercise.

Note

A pointer can be named using any legal variable name. Some programmers use names that make it clear the variable is a pointer, such as ending the name with **ptr** *or beginning the name with* **p_**, *but it is not necessary.*

The Hexadecimal Number System

Hexadecimal numbers (often called just "hex") are ideal for representing memory locations. The hexadecimal number system is a base 16 number system. Because base 16 is a multiple of base 2 (binary), the hexadecimal number system is compatible with the computer's internal representations. Hexadecimal numbers can display large values in a form that is easier to read than binary numbers.

In Exercise 10-1, you declared the pointer (**int *int_ptr;**) and initialized it in a different statement (**int_ptr = &i;**). Like other variables, you can initialize a pointer when you declare it. For example, the statement below could have been used in the program in Exercise 10-1 to declare the pointer and initialize it in one statement.

```
int *int_ptr = &i;
```

USING THE * AND & OPERATORS

The dereferencing operator (*) is used for more than declaring pointers. In the statement below, the dereferencing operator tells the compiler to return the value in the variable being pointed to, rather than the address of the variable.

```
result = *int_ptr;
```

The variable **result** is assigned the value of the integer pointed to by **int_ptr**.

EXERCISE 10-2 THE * AND & OPERATORS

1. Enter the following lines of code to the program you saved named *POINTER.CPP*:

```
cout << "&i = " << &i << '\n';
cout << "*int_ptr = " << *int_ptr << '\n';
```

The output statement with the **&i** does the same thing as outputting **int_ptr**, since **int_ptr** holds the address of **i.** Sending ***int_ptr** to the output stream prints the contents of the variable pointed to by **int_ptr**, rather than printing the pointer itself.

2. Compile and run to see the output of the new statements.
3. Enter the following statements at the end of the program:

```
int_ptr = &j; // store the address of j to int_ptr
cout << "int_ptr = " << int_ptr << '\n';
cout << "*int_ptr = " << *int_ptr << '\n';
```

4. Compile and run again. Because **int_ptr** now points to the integer **j** rather than **i**, the output statement prints the value of **j**, even though the exact statement (**cout << "*int_ptr = " << *int_ptr << '\n';**) printed the value of **i** just two statements back.
5. Enter the following statements at the end of the program:

```
int_ptr = &k; // store the address of k to int_ptr
cout << "int_ptr = " << int_ptr << '\n';
cout << "*int_ptr = " << *int_ptr << '\n';
```

6. Compile and run again. The pointer now points to the variable **k**, so ***int_ptr** returns the value of **k**, which is 3.
7. Save and close the source code file.

CHANGING VALUES WITH *

The dereferencing operator allows you to do more than get the value in the variable the pointer is pointing to. You can change the value of the variable the pointer points to. For example, the statement below assigns the value 5 to the integer to which **int_ptr** points.

```
*int_ptr = 5;
```

You are probably now beginning to see why C++ is so powerful—and so dangerous. A statement like the one above should be avoided because it fails to indicate the specific variable that is being changed. Although C++ programmers are given the freedom to work with memory in a rich variety of ways, pointers should be used only when they improve the program.

Consider the program in Figure 10-3. The program declares and initializes two floating-point numbers. Using a do while loop, the program repeatedly picks the larger of the two numbers and divides it by 2. The loop ends when one of the variables becomes less than 1.

The program uses a pointer to provide an efficient solution. By setting a pointer to point to the larger of the two values, the larger value can be printed and divided by 2 by use of the pointer, rather than the variable itself. By using a pointer, the same code can be used no matter which variable is the larger.

```
#include <iostream.h> // necessary for cout command

int main()
{
  float a, b;          // declare two floating-point numbers
  float *float_ptr;    // declare a pointer to a float

  a = 169.8;
  b = 237.5;

  do
   {
     cout << "The two numbers are " << a << " and " << b << endl;

     if (a >= b)
      {
       float_ptr = &a;
      }
     else
      {
       float_ptr = &b;
      }

     cout << "The largest of the two numbers is "
          << *float_ptr << endl;
     cout << *float_ptr;
     *float_ptr = *float_ptr / 2;
     cout << " divided by 2 is " << *float_ptr << endl;
  } while((a > 1.0) && (b > 1.0));
  return 0;
}
```

FIGURE 10-3
This program accesses variables by use of a pointer.

EXERCISE 10-3 USING POINTERS

1. Enter the program shown in Figure 10-3. Save the source code file as *THEPOINT.CPP*.
2. Study the source code closely before you run the program. Compile and run the program to see its output. The program divides the larger of the 2 values by 2 until one of the values becomes less than 1.
3. Close the source code file.

On the Net

Learn more about pointers and see more examples at http://www.ProgramCPP.com. See topic 10.1.1.

SECTION 10.1 QUESTIONS

1. What is stored in a pointer?
2. What symbol is used for the address-of operator?
3. Write a statement that declares a pointer to a variable of type double.
4. Write a statement that assigns the pointer you declared in question 3 to the address of a variable named x.
5. Now change the value of the variable x to 9.9 using the pointer you declared and the dereferencing operator.

PROBLEM 10.1.1

Write a program that declares two variables of type float and a single pointer of type float. First initialize the pointer to point to the first float variable. Use the dereferencing operator to initialize the variable to 1.25. Next point the pointer to the second float variable and use the pointer to initialize the variable to 2.5. Print the value of the first variable to the screen by accessing the variable directly. Print the second variable using the dereferencing operator. Save the source code file as *POINTER2.CPP*.

CHAPTER 10, SECTION 2

More About Character Arrays

Now that you have been exposed to the basics of pointers, some matters relating to character arrays may make more sense. In this section, you will strengthen your knowledge about character arrays.

ALL CHARACTER ARRAYS USE A POINTER

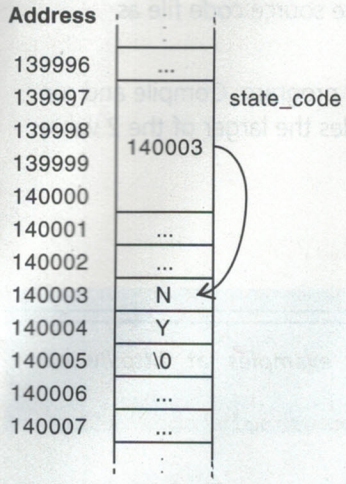

FIGURE 10-4
The name of a character array is actually a pointer constant.

The first example of a pointer in this chapter was a pointer to a character array. Let's look at that figure again (see Figure 10-4).

When you declare a character array, the name you give the array is actually a ***pointer constant.*** A pointer constant is like a ***pointer variable,*** except you cannot change what it points to. In Figure 10-4, the **state_code** pointer constant is initialized by the compiler to point to whatever location is assigned to the first character of the character array.

Note

If you were wondering why C++ requires you to use the **strcpy** *function to assign a string to a character array, here is why. A statement like the one below would try to assign a string of characters to a pointer constant.*

```
state_code = "NY"; // CAN'T DO THIS!
```

USING SUBSCRIPT NOTATION

C++ allows you to access any character in a character array individually using a method called ***subscript notation.*** Subscript notation looks like the syntax you use when declaring a character array. Consider the program in Figure 10-5. The character array **my_word** is declared to be five characters long and assigned the word *book*.

```
#include <iostream.h> // necessary for cout command

int main()
{
  char my_word[5] = "book";
  cout << my_word << '\n';
  return 0;
}
```

FIGURE 10-5
In this program, the character array **my_word** is initialized with the word *book*.

Recall from Chapter 6 that the characters of a character array are stored together in memory. What you may not know, however, is that the characters of the array are numbered, beginning with zero (see Figure 10-6). So a five-character array is numbered 0 through 4.

Using subscript notation, you can change the first character in the array using a statement like the one below.

```
my_word[0] = 't';
```

The first character is changed to a 't' because the first character is at position 0. You may wonder why the people who created C++ made the subscript of the

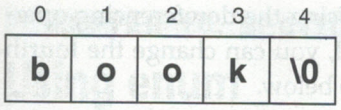

FIGURE 10-6
The characters of an array are numbered, beginning with zero.

first character 0 rather than 1. You might find it helpful to think of the number in the brackets as the number of places you must move over from the first character in the array.

The first character in the array is where the pointer points. To access the fourth character, however, you must move three times. An exercise will give you a little practice with subscript notation.

EXERCISE 10-4 SUBSCRIPT NOTATION

1. Enter the program below. Save the source code as *WORDPLAY.CPP*.

```
#include <iostream.h> // necessary for cout command

int main()
{
  char my_word[5] = "book";

  cout << my_word << '\n';

  my_word[3] = 't';
  cout << my_word << '\n';
  return 0;
}
```

2. Compile and run the program to see how subscript notation changed one character in the word.
3. Add the following statements to the bottom of the program.

```
  my_word[0] = 's';
  cout << my_word << '\n';
```

4. Run the program again to see the effect of the statements you added.
5. Save the source code and leave it open for the next exercise.

PITFALLS

Be careful when accessing characters in an array individually. The compiler will not prevent you from going beyond the length of your array. You could end up changing other data in memory instead of your array. If the null terminator of your string is changed to another character, the compiler will allow it and errors will result.

Subscript notation can be used without knowledge of pointers. However, since you are becoming familiar with pointers, you would probably be interested in seeing how to change a character in the array without subscript notation.

USING THE * OPERATOR IN CHARACTER ARRAYS

You already know that **my_word** is a pointer to the first character in the array. Because characters occupy one byte of memory, it makes sense that the second

character of the array is stored at `my_word + 1`. Using the dereferencing operator and this knowledge of how the array is stored, you can change the fourth character in the array using a statement like the one below.

```
*(my_word + 3) = 'n';  // has the same result as my_word[3] = 'n';
```

You can see that subscript notation makes for more readable code than the dereferencing operator. But as statements like the one above begin to make sense to you, you will begin to unlock the real power of C++. Let's add the statement to the program you entered in Exercise 10-4.

EXERCISE 10-5 CHANGING ARRAY CHARACTERS

1. Add the following lines to the bottom of the program on your screen.

```
*(my_word + 3) = 'n';  // has the same result as my_word[3] = 'n';
cout << my_word << '\n';
```

2. Run the program to see the effect of the new statements.
3. Add a statement that uses subscript notation to change the word to *sown*. Add an output statement to output the new word.
4. Run, save, and close.

SECTION 10.2 QUESTIONS

1. The name of a character array is what kind of pointer?
2. Why can't you assign a string to a character array using the assignment operator?
3. What is the name of the method that allows you to access individual characters of a character array using brackets ([])?
4. Given the character array declared as `char A[8] = "ABCDEFG";`, what character is returned by `A[2]`?
5. Using the same character array you used for question 4, what would be the resulting string if the following statement were executed?

```
A[1] = 'X';
```

PROBLEM 10.2.1

Write a program that declares a character array named `alphabet` and initializes the array to "ABCDEFGHIJKLMNOPQRSTUVWXYZ." In your program, include a loop that replaces one character of the array at a time with the lowercase letters a–z. Print the character array to the screen during each iteration of the loop. *Hint:* Remember that a character array is an array of integer values. Use ASCII values to make the changes. Save the source code file as *ALPHABET.CPP*.

Using enum

The *enum* keyword (short for enumerated) is a C++ feature that is often overlooked. It allows you to create your own simple data types for special purposes in your program. For example, you could create a data type called colors that allows only the values *red*, *green*, *blue*, and *yellow* as data. In this section, you will learn how enum works and how you can use it in your programs.

HOW TO USE enum

The enum keyword is easy to use. You simply create a type, give it a name, and tell the compiler what values your new data type will accept. Consider the statement below.

```
enum sizes {small, medium, large, jumbo};
```

The data type called **sizes** can have one of four values: *small*, *medium*, *large*, or *jumbo*. The next step is to declare a variable that uses **sizes** as a type. Let's declare two variables of the type **sizes**.

```
enum sizes drink_size, popcorn_size;
```

The variable **drink_size** and **popcorn_size** are of type **sizes** and can be assigned one of the four sizes defined in the **sizes** type.

EXERCISE 10-6 USING enum

1. Enter the program below. Save the source code as *ENUMTEST.CPP*.

```
#include <iostream.h> // necessary for cout command

int main()
{
  enum sizes {small, medium, large, jumbo};
  sizes drink_size, popcorn_size;

  drink_size = large;
  popcorn_size = jumbo;

  if (drink_size == large)
   { cout << "You could have a jumbo for another quarter.\n"; }

  if ((popcorn_size == jumbo) && (drink_size != jumbo))
   { cout << "You need more drink to wash down a jumbo popcorn.\n"; }
  return 0;
}
```

2. Run the program to see the output. Close the source code file.

HOW enum WORKS

Internally, the compiler assigns an integer to each of the items in an enum list. For example, the statement below does not print small, medium, large, or jumbo to the screen. It prints either 0, 1, 2, or 3.

```
cout << drink_size << '\n';
```

WARNING

Attempting to print the value of an enumerated type results in an error on some compilers. Enumerated types are best used in expressions and switch structures, rather than directly for output.

By default, the compiler begins assigning integers with zero. For example, in the sizes type, small = 0, medium = 1, large = 2, and jumbo = 3. You can, however, choose your own values. For example, suppose you wanted to use an enumerated type to assign quantities. You could use a statement like the one below to declare an enumerated type with the values 1, 2, 12, 48, and 144.

```
enum quantity {Single=1, Dozen=12, Full_Case=48, Gross=144};
```

As another example, suppose you want to create a type called month that is made up of the months of the year. Because the months are commonly numbered 1 through 12, you decide to have the compiler assign those numbers to your list. The assignment in the first value of the list sets the beginning value for the items in the list, as in the statement below. January will be assigned the value 1, February the value 2, etc.

```
enum month {January=1, February, March, April, May, June, July,
            August, September, October, November, December};
```

You can use the fact that enum uses integers to your advantage. For example, a statement like the one below can be used with the sizes type.

```
if (drink_size > medium)
  { cout << "This drink will not fit in your cup holder.\n"; }
```

USING typedef

Another C++ feature which is related to enum is typedef. You can use typedef to give a new name to an existing data type. For example, if you prefer to use the term *real* to declare variable of type **float**, you can give the data type an *alias* of real with typedef.

```
typedef float real;
```

You should, of course, use typedef sparingly because you may confuse the reader of your code.

You can use typedef for more than just changing the names of data types to fit your liking. For example, recall from Chapter 4 that some C++ compilers

include a boolean data type and some compilers do not. You can use typedef and a couple of constants to create your own boolean data type.

```
typedef int bool;
const int TRUE = 1;
const int FALSE = 0;
```

The three statements above make it possible to declare variables of type **bool** and assign values to the variables using **TRUE** and **FALSE**.

```
bool acceptable;
acceptable = TRUE;
```

On the Net

For more information about typedef, including how typedef can be used to make code easier to move among compilers, see http://www.ProgramCPP.com. See topic 10.3.1.

SECTION 10.3 QUESTIONS

1. Write a statement that declares an enum type called speed that allows the values *stopped*, *slow*, and *fast*.
2. Write a statement that declares a variable named rabbit of the type you declared in question 1.
3. Write a statement that assigns the value fast to the *rabbit* variable you declared above.
4. What does the compiler use internally to represent the values of an enum type?
5. Write a statement that declares an enum type called temperature that allows the values *frigid*, *cold*, *cool*, *mild*, *warm*, *hot*, and *sizzling*. Have the list begin with the value 1.
6. What can be used to give a new name to an existing data type?

PROBLEM 10.3.1

Make a list of several different enumerated data types that could be useful in programs. For example, **enum TrueFalse { FALSE, TRUE}** or **enum Position {open, closed}**.

CHAPTER 10, SECTION 4

Structures

C++ ***structures*** allow variables to be grouped to form a new data type. The data elements in a structure are arranged in a manner that is similar to the way database programs arrange data.

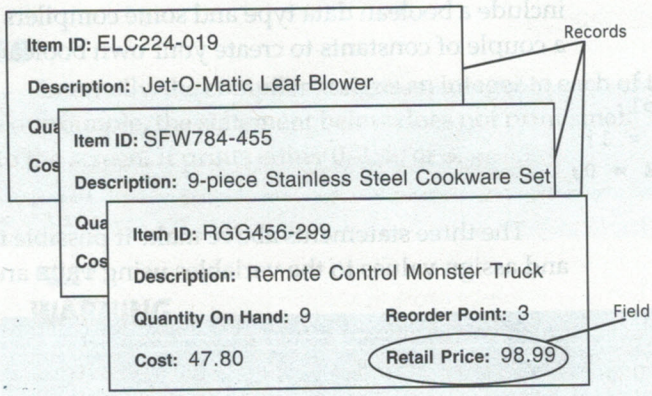

FIGURE 10-7
A record in a database is one completed set of fields.

Note

All data in a C++ program is stored in a ***data structure****. Any organized way of storing data in a computer is a data structure. The basic variable types such as* `int` *and* `float` *are called* ***primitive data structures****. A character array is an example of a category of data structures called* ***simple data structures****. The term structure used in this section refers to a specific data structure made by grouping other data structures. Do not confuse the term* structure *used in this section with the more generic term* data structure.

In a database program, data is stored in ***records.*** For example, suppose you have a database of items sold by a mail-order company. Each item that the company sells is stored as a record in the database. Each record is made up of data called ***fields.*** Figure 10-7 shows a series of three records contained in a database. Notice that each record has identical field names (i.e., Item ID).

C++ allows you to create a record by grouping the variables and arrays necessary for the fields into a single structure. The variables in the structure can be of mixed types. For example, in Figure 10-7, character arrays must be used for the item ID and the description, an integer type can be used to store the quantity on hand and reorder point, and a floating-point type is necessary for cost and retail price.

DECLARING AND USING STRUCTURES

A structure must be declared. Because a structure is made up of more than one variable, a special syntax is used to access the individual variables of a structure.

DECLARING A STRUCTURE

Shown below is the declaration for the structure in our example.

```
struct inventory_item
  {
    char item_ID[11];
    char description[31];
    int quantity_on_hand;
    int reorder_point;
    float cost;
```

```
    float retail_price;
};
```

The **struct** keyword identifies the declaration as a structure. The identifier associated with the structure is **inventory_item**. The variables in the structure are called *members*. The members of the structure are placed within braces. Within the braces, however, the variables and arrays are declared using the syntax to which you are accustomed.

Once you have declared a structure, you must declare a variable that is of the structure's type. This may seem confusing, but what the struct key word does is define a new data type. You can then create as many variables as you want of the new type. The statement below creates a variable named **todays_special** that is of type **inventory_item**.

```
inventory_item todays_special;
```

ACCESSING MEMBERS OF A STRUCTURE

Accessing data in a structure is surprisingly simple. To access a member of the structure, use the name of the variable, a period (.), then the name of the member you need to access, as shown in Figure 10-8. The period is actually an operator called the *dot operator.*

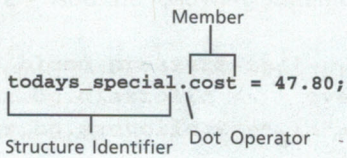

FIGURE 10-8
To access a member of a structure, use the name of the variable, a period, and the name of the member you need to access.

The code segment below declares a variable named **todays_special** of the type **inventory_item** and initializes each member of the structure.

```
inventory_item todays_special;

strcpy(todays_special.item_ID, "RGG456-299");
strcpy(todays_special.description, "Remote Control Monster Truck");
todays_special.quantity_on_hand = 19;
todays_special.reorder_point = 3;
todays_special.cost = 47.80;
todays_special.retail_price = 98.99;
```

EXERCISE 10-7 · STRUCTURES

1. Retrieve the source code file *STRUCT.CPP*. A program appears that includes the declaration and initialization of the **todays_special** structure variable.
2. Enter the following code at the bottom of the program (before the closing brace, of course).

```
cout << "Today's Special\n";
cout << "      Item ID: " << todays_special.item_ID << endl;
```

```
cout << "  Description: " << todays_special.description << endl;
cout << "     Quantity: " << todays_special.quantity_on_hand << endl;
cout << "Regular Price: " << setprecision(2)
     << todays_special.retail_price << endl;
cout << "   Sale Price: " << todays_special.retail_price * 0.8
     << endl;
```

3. Compile and run the program to see the output from the structure.
4. Save the source code file and close.

NESTED STRUCTURES

A structure can include enumerated data types and even other structures as members. Consider the program in Figure 10-9. The program sets up a structure to be used to store vital data about blood donors. The structure named **donor_info** includes two enumerated data types (**blood_type** and **rh_factor**) and a structure (**blood_pressure**) among its members.

```
enum blood_type { unknown, A, B, AB, O };
enum rh_factor { negative, positive };

struct blood_pressure
  {
   int systolic;
   int diastolic;
  };

struct donor_info
  {
   blood_type type;
   rh_factor rh;
   blood_pressure bp;
   int heart_rate;
  };

int main()
{
  donor_info current_donor;

  current_donor.type = A;
  current_donor.rh = positive;
  current_donor.bp.systolic = 130;
  current_donor.bp.diastolic = 74;
  current_donor.heart_rate = 69;
  return 0;
}
```

FIGURE 10-9
This program uses nested structures to store blood pressure values.

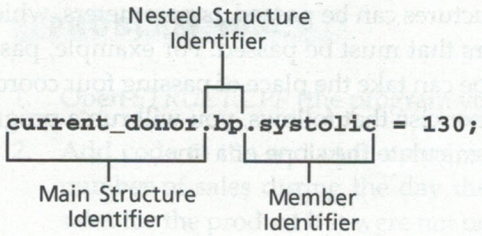

FIGURE 10-10
This assignment stores the value 130 in the systolic variable which is in the `bp` structure which is in the `current_donor` structure.

The **blood_type** and **rh_factor** data are good candidates for an enumerated data type because only a few values are possible. Because blood pressure is actually two values, a structure is used to group the two values into one variable. When a structure appears within a structure the resulting data structure is called a ***nested structure.***

Accessing the nested structure requires that two periods be used. As the statement in Figure 10-10 illustrates, initializing the blood pressure values requires that the first structure be accessed by name, then the structure variable within the first structure, and finally, the variable within the nested structure.

EXERCISE 10-8 NESTED STRUCTURES

1. Enter the program shown in Figure 10-9. Save the source code as *DONORS.CPP*.
2. Add the following statement to the main function.

```
cout << "The donor's blood pressure is "
     << current_donor.bp.diastolic << " over "
     << current_donor.bp.systolic << ".\n";
```

3. Add the appropriate directive to include the code necessary for the cout statements.
4. Compile and run the progra1m. Save and close the source code.

It is easy to get locked into thinking about structures in terms of database applications. Structures, however, have many other applications. For example, some mathematical or graphical applications use coordinates such as (x,y) in calculations. You can use a structure like the one below to group the x and y into a structure that represents a graphical point.

```
struct point
  {
   float x;
   float y;
  };
```

You might then want to use nested structures to create a data type that defines two points, that when connected, form a line, as shown below.

```
struct line
  {
   point p1;
   point p2;
  };
```

Structures can be passed as parameters, which can reduce the number of parameters that must be passed. For example, passing a variable of the line structure type can take the place of passing four coordinates (two x and two y values). In the exercise that follows, you will run a program that passes a structure variable to calculate the slope of a line.

EXERCISE 10-9 SLOPE OF A LINE

1. Open *LNSLOPE.CPP*. A program appears that uses the structures above to calculate the slope of a line.
2. Compile and run the program. Provide the points (1,2) and (2,5) as input.
3. Add the following statements after the declaration of the variable **m**.

```
line horizontal_line;

horizontal_line.p1.x = 1;
horizontal_line.p1.y = 2;
horizontal_line.p2.x = 5;
horizontal_line.p2.y = 2;
m = slope(horizontal_line);
cout << "The slope of every horizontal line is " << m << endl;
```

4. Compile and run the program again to see the result of the new lines.
5. Save the source code and close.

SECTION 10.4 QUESTIONS

1. What is the purpose of a structure?
2. What is the term for the variables in a structure?
3. Write a declaration for a structure named **house** that is to be used to store records in a database of house descriptions for a real estate agent. The fields in the record should include address, square footage of the house, number of bedrooms, number of bathrooms, number of cars that can fit in the garage, and the listed price of the house.
4. Write a statement that declares a structure variable named **featured_home** using the structure declared in question 3.
5. Write a series of statements that initialize the structure variable declared in question 4. Do your best to initialize the structure variable with realistic values.
6. Write a statement that prints the information in the **featured_home** structure variable in the format of the example below, where 3-2-2 is the number of bedrooms, baths, and garage stalls respectively, and 1800 is the square footage.

   3918 Shonle Road 3-2-2 1800 $64,000

PROBLEM 10.4.1

1. Open *STRUCT.CPP* (the program you saved in Exercise 10-7).
2. Add code to the end of the program that asks the user for the anticipated number of sales during the day the item is on special and the anticipated sales for the product if it were not on special.
3. Check the value entered against the quantity on hand to make sure that the anticipated orders can be filled. Warn the user if the quantity on hand is less than the anticipated sales.
4. Calculate the amount of income the product is anticipated to generate if the product is put on special and the amount of income the product is anticipated to generate if the product is not put on special.
5. Save the new source code as *STRUCT2.CPP*. Compile and run the program. After testing the program, close the source code file.

PROBLEM 10.4.2

Write a program that uses the point and line structures from Exercise 10-9 to calculate the midpoint of a given line. Have the program ask the user for the points that define the line, then use the formula below to calculate the midpoint of the line and output the coordinates of the midpoint. Save the program as *MIDPOINT.CPP*.

$$\left(\frac{x_1 + x_2}{2}, \frac{y_1 + y_2}{2}\right)$$

KEY TERMS

address-of operator	pointer
data structure	pointer constant
dereferencing operator	pointer variable
dot operator	primitive data structure
enum	simple data structure
fields	records
members	structures
nested structure	subscript notation

SUMMARY

- Pointers are variables and constants that hold memory addresses.
- The dereferencing operator (*) is used to declare pointers and to access the value in the variable to which the pointer points.

- The address-of operator (&) returns the address of a variable, rather than the value in the variable.
- The name you give a character array is actually a pointer constant that points to the first character in the array.
- Subscript notation is a method of accessing individual characters in a character array. You can also access individual characters in an array using the dereferencing operator and adding the correct number to the pointer.
- The enum keyword allows you to create custom data types. Internally, the values you include in your enum data types are stored as integers.
- Structures are very useful data structures that allow variables to be grouped to form a new data type.
- The variables within a structure are called members.

PROJECTS

PROJECT 10-1 • REVERSING A STRING

Write a function named **`reverse_string`** that reverses a string. For example, the function would change the string "ABCDEF" to "FEDCBA." The function should not reverse the position of the null terminator. Write a main function to test the **`reverse_string`** function.

PROJECT 10-2 • CRYPTOGRAPHY

Write a program that uses a loop to encrypt a string by adding 1 to the ASCII value of each character in the string. The string "ABCDEF" would become "BCDEFG." The string "apple" would become "bqqmf."

PROJECT 10-3 • CRYPTOGRAPHY

Modify the program from Project 10-2 to decrypt a string that has been encrypted using the method of Project 10-2.

PROJECT 10-4 • GEOMETRY

On a coordinate plane, the length of a line connecting two points can be found using the formula below.

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Extend the *LNSLOPE.CPP* program you saved in Exercise 10-9 to calculate the length of the line using the formula above.

PROJECT 10-5 • GEOMETRY

Write a program that accepts two points as input. Have the program calculate a third point that, with the other two points in a plane, form a right triangle.