Arrays of Structures

This chapter builds on the previous one by showing you how to create many structures for your data. After creating an array of structures, you can store many occurrences of your data values.

Arrays of structures are good for storing a complete employee file, inventory file, or any other set of data that fits in the structure format. Whereas arrays provide a handy way to store several values that are the same type, arrays of structures store several values of different types together, grouped as structures.

This chapter introduces the following concepts:

- Creating arrays of structures
- Initializing arrays of structures
- Referencing elements from a structure array
- Arrays as members

Many C++ programmers use arrays of structures as a prelude to storing their data in a disk file. You can input and calculate your disk data in arrays of structures, and then store those structures in memory. Arrays of structures also provide a means of holding data you read from the disk.

Declaring Arrays of Structures

It is easy to declare an array of structures. Specify the number of reserved structures inside array brackets when you declare the structure variable. Consider the following structure definition:

```
struct stores
     { int employees;
          int registers;
          double sales;
     } store1, store2, store3, store4, store5;
```

This structure should not be difficult for you to understand because there are no new commands used in the structure declaration. This structure declaration creates five structure variables. Figure 29.1 shows how C++ stores these five structures in memory. Each of the structure variables has three members—two integers followed by a double floating-point value.

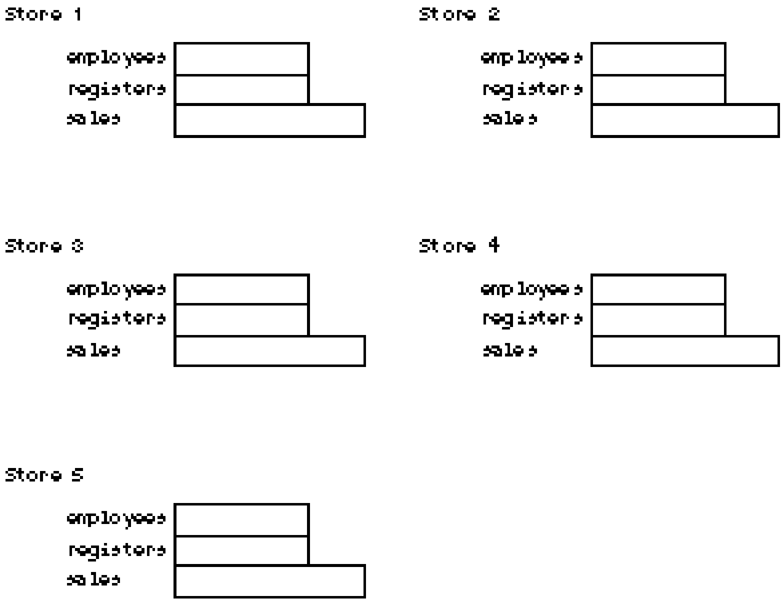


Figure 29.1. The structure of `Store 1`, `Store 2`, `Store 3`, `Store 4`, and `Store 5`.

If the fourth store increased its employee count by three, you could update the store's employee number with the following assignment statement:

```
store4.employees += 3;          // Add three to this store's
                                // employee count.
```

Suppose the fifth store just opened and you want to initialize its members with data. If the stores are a chain and the new store is similar to one of the others, you can begin initializing the store's data by assigning each of its members the same data as another store's, like this:

```
store5 = store2;              // Define initial values for
                              // the members of store5.
```

Arrays of structures make working with large numbers of structure variables manageable.

Such structure declarations are fine for a small number of structures, but if the stores were a national chain, five structure variables would not be enough. Suppose there were 1000 stores. You would not want to create 1000 different store variables and work with each one individually. It would be much easier to create an array of store structures.

Consider the following structure declaration:

```
struct stores
     { int employees;
          int registers;
          double sales;
     } store[1000];
```

In one quick declaration, this code creates 1000 store structures, each one containing three members. Figure 29.2 shows how these structure variables appear in memory. Notice the name of each individual structure variable: `store[0]`, `store[1]`, `store[2]`, and so on.

CAUTION: Be careful that your computer does not run out of memory when you create a large number of structures. Arrays of structures quickly consume valuable memory. You might have to create fewer structures, storing more data in disk files and less data in memory.

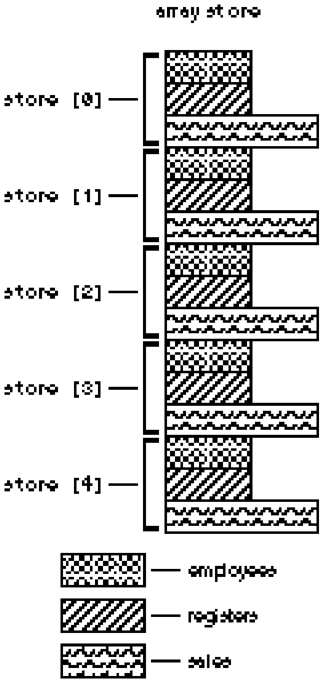


Figure 29.2. An array of the `store` structures.

The element `store[2]` is an array element. This array element, unlike the others you have seen, is a structure variable. Therefore, it contains three members, each of which you can reference with the dot operator.

The dot operator works the same way for structure array elements as it does for regular structure variables. If the number of employees for the fifth store (`store[4]`) increased by three, you could update the structure variable like this:

```
store[4].employees += 3;        // Add three to this store's
                                // employee count.
```

You can assign complete structures to one another also by using array notation. To assign all the members of the 20th store to the 45th store, you would do this:

```
store[44] = store[19];     // Copy all members from the
                           // 20th store to the 45th.
```

The rules of arrays are still in force here. Each element of the array called `store` is the same data type. The data type of `store` is the structure `stores`. As with any array, each element must be the same data type; you cannot mix data types in the same array. This array's data type happens to be a structure you created containing three members. The data type for `store[316]` is the same for `store[981]` and `store[74]`.

The name of the array, `store`, is a pointer constant to the starting element of the array, `store[0]`. Therefore, you can use pointer notation to reference the stores. To assign `store[60]` the same value as `store[23]`, you can reference the two elements like this:

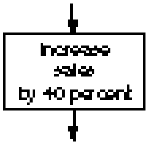
```
*(store+60) = *(store+23);
```

You also can mix array and pointer notation, such as

```
store[60] = *(store+23);
```

and receive the same results.

You can increase the sales of `store[8]` by 40 percent using pointer or subscript notation as well, as in



```
store[8].sales = (*(store+8)).sales * 1.40;
```

The extra pair of parentheses are required because the dot operator has precedence over the dereferencing symbol in C++'s hierarchy of operators (see Appendix D, "C++ Precedence Table"). Of course, in this case, the code is not helped by the pointer notation. The following is a much clearer way to increase the `sales` by 40 percent:

```
store[8].sales *= 1.40;
```

The following examples build an inventory data-entry system for a mail-order firm using an array of structures. There is very little new you have to know when working with arrays of structures. To become comfortable with the arrays of structure notation, concentrate on the notation used when accessing arrays of structures and their members.

Keep Your Array Notation Straight

You would never access the member `sales` like this:

```
store.sales[8] = 3234.54;          // Invalid
```

Array subscripts follow only array elements. `sales` is not an array; it was declared as being a double floating-point number. `store` can never be used without a subscript (unless you are using pointer notation).

Here is a corrected version of the previous assignment statement:

```
store[8].sales=3234.54;            // Correctly assigns
                                   // the value.
```

Examples

1. Suppose you work for a mail-order company that sells disk drives. You are given the task of writing a tracking program for the 125 different drives you sell. You must keep track of the following information:

   Storage capacity in megabytes
   Access time in milliseconds
   Vendor code (A, B, C, or D)
   Cost
   Price

   Because there are 125 different disk drives in the inventory, the data fits nicely into an array of structures. Each array element is a structure containing the five members described in the list.

   The following structure definition defines the inventory:

```
struct inventory
{
     long int storage;
     int access_time;
     char vendor_code;
     double code;
     double price;
} drive[125];  // Defines 125 occurrences of the structure.
```

2. When working with a large array of structures, your first concern should be how the data inputs into the array elements. The best method of data-entry depends on the application.

   For example, if you are converting from an older computerized inventory system, you have to write a conversion program that reads the inventory file in its native format and saves it to a new file in the format required by your C++ programs. This is no easy task. It demands that you have extensive knowledge of the system from which you are converting.

   If you are writing a computerized inventory system for the first time, your job is a little easier because you do not have to convert the old files. You still must realize that someone has to type the data into the computer. You must write a data-entry program that receives each inventory item from the keyboard and saves it to a disk file. You should give the user a chance to edit inventory data to correct any data he or she originally might have typed incorrectly.

   One of the reasons disk files are introduced in the last half of the book is that disk-file formats and structures share a common bond. When you store data in a structure, or more often, in an array of structures, you can easily write that data to a disk file using straightforward disk I/O commands.

   The following program takes the array of disk drive structures shown in the previous example and adds a data-entry function so the user can enter data into the array of structures. The program is menu-driven. The user has a choice, when starting the program, to add data, print data onscreen, or exit the program. Because you have yet to see disk I/O commands, the data in the array of structures goes away when the program ends. As mentioned earlier, saving those structures to disk is an easy task after you learn C++'s disk I/O commands. For now, concentrate on the manipulation of the structures.

This program is longer than many you previously have seen in this book, but if you have followed the discussions of structures and the dot operator, you should have little trouble following the code.

Identify the program and include the necessary header files. Define a structure that describes the format of each inventory item. Create an array of structures called `disk`*.*

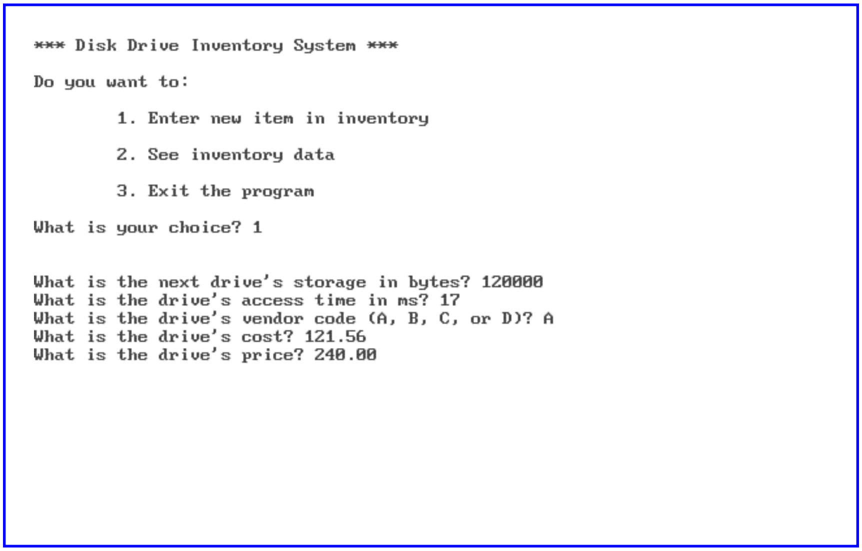
Display a menu that gives the user the choice of entering new inventory data, displaying the data on-screen, or quitting the program. If the user wants to enter new inventory items, prompt the user for each item and store the data into the array of structures. If the user wants to see the inventory, loop through each inventory item in the array, displaying each one on-screen.

```
// Filename: C29DSINV.CPP
// Data-entry program for a disk drive company.
#include <iostream.h>
#include <stdlib.h>
#include <iomanip.h>
#include <stdio.h>

struct inventory            // Global structure definition.
{
   long int storage;
   int access_time;
   char vendor_code;
   float cost;
   float price;
};                 // No structure variables defined globally.

void disp_menu(void);
struct inventory enter_data();
void see_data(inventory disk[125], int num_items);

void main()
```

```
{
   inventory disk[125];     // Local array of structures.
   int ans;
   int num_items=0;                   // Number of total items
                                      // in the inventory.

   do
     {
        do
         { disp_menu();      // Display menu of user choices.
           cin >> ans;                 // Get user's request.
         } while ((ans<1) || (ans>3));

         switch (ans)
      { case (1): { disk[num_items] = enter_data(); // Enter
                                                  // disk data.
                    num_items++;    // Increment number of items.
                    break; }
        case (2): { see_data(disk, num_items);   // Display
                                                  // disk data.
                    break; }
        default : { break; }
      }
      }  while (ans!=3);               // Quit program
                                       // when user is done.
     return;
}

void disp_menu(void)
{

   cout << "\n\n*** Disk Drive Inventory System ***\n\n";
   cout << "Do you want to:\n\n";
   cout << "\t1. Enter new item in inventory\n\n";
   cout << "\t2. See inventory data\n\n";
   cout << "\t3. Exit the program\n\n";
   cout << "What is your choice? ";
   return;
}

inventory enter_data()
```

```
{
   inventory disk_item;    // Local variable to fill
                           // with input.

   cout << "\n\nWhat is the next drive's storage in bytes? ";
   cin >> disk_item.storage;
   cout << "What is the drive's access time in ms? ";
   cin >> disk_item.access_time;
   cout << "What is the drive's vendor code (A, B, C, or D)? ";
   fflush(stdin);  // Discard input buffer
                   // before accepting character.
   disk_item.vendor_code = getchar();
   getchar();  // Discard carriage return
   cout << "What is the drive's cost? ";
   cin >> disk_item.cost;
   cout << "What is the drive's price? ";
   cin >> disk_item.price;

   return (disk_item);
}

void see_data(inventory disk[125], int num_items)
{
   int ctr;
   cout << "\n\nHere is the inventory listing:\n\n";
   for (ctr=0;ctr<num_items;ctr++)
      {
      cout << "Storage: " << disk[ctr].storage << "\t";
      cout << "Access time: " << disk[ctr].access_time << "\n";
      cout << "Vendor code: " << disk[ctr].vendor_code << "\t";
      cout << setprecision(2);
      cout << "Cost: $" << disk[ctr].cost << "\t";
      cout << "Price: $" << disk[ctr].price << "\n";
      }
   return;

}
```

Figure 29.3 shows an item being entered into the inventory file. Figure 29.4 shows the inventory listing being displayed to the screen. There are many features and error-checking functions you can add, but this program is the foundation of a more comprehensive inventory system. You can easily adapt it to a different type of inventory, a video tape collection, a coin collection, or any other tracking system by changing the structure definition and the member names throughout the program.



```
*** Disk Drive Inventory System ***

Do you want to:

        1. Enter new item in inventory

        2. See inventory data

        3. Exit the program

What is your choice? 1


What is the next drive's storage in bytes? 120000
What is the drive's access time in ms? 17
What is the drive's vendor code (A, B, C, or D)? A
What is the drive's cost? 121.56
What is the drive's price? 240.00
```

Figure 29.3. Entering inventory information.

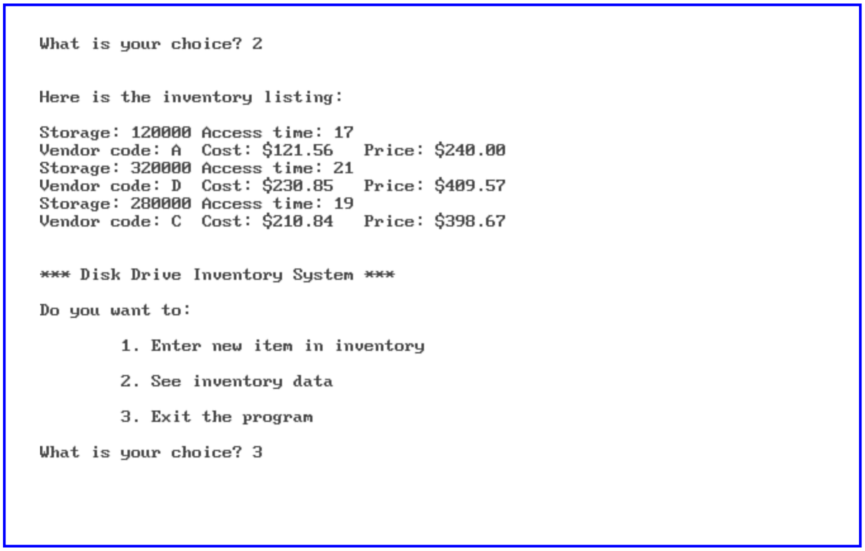
Arrays as Members

Members of structures can be arrays. Array members pose no new problems, but you have to be careful when you access individual array elements. Keeping track of arrays of structures that contain array members might seem like a great deal of work on your part, but there is nothing to it.

Consider the following structure definition. This statement declares an array of 100 structures, each structure holding payroll information for a company. Two of the members, `name` and `department`, are arrays.

```
struct payroll
   { char name[25];                // Employee name array.
```

```
    int dependents;
    char department[10];        // Department name array.
    float salary;
  } employee[100];            // An array of 100 employees.
```



```
What is your choice? 2


Here is the inventory listing:

Storage: 120000 Access time: 17
Vendor code: A  Cost: $121.56   Price: $240.00
Storage: 320000 Access time: 21
Vendor code: D  Cost: $230.85   Price: $409.57
Storage: 280000 Access time: 19
Vendor code: C  Cost: $210.84   Price: $398.67


*** Disk Drive Inventory System ***

Do you want to:

        1. Enter new item in inventory

        2. See inventory data

        3. Exit the program

What is your choice? 3
```

Figure 29.4. Displaying the inventory data.

Figure 29.5 shows what these structures look like. The first and third members are arrays. `name` is an array of 25 characters, and `department` is an array of 10 characters.

Suppose you must save the 25th employee's initial in a character variable. Assuming initial is already declared as a character variable, the following statement assigns the employee's initial to the varible `initial`:

```
initial = employee[24].name[0];
```

The double subscripts might look confusing, but the dot operator requires a structure variable on its left (`employee[24]`) and a member on its right (`name`'s first array element). Being able to refer to member arrays makes the processing of character data in structures simple.

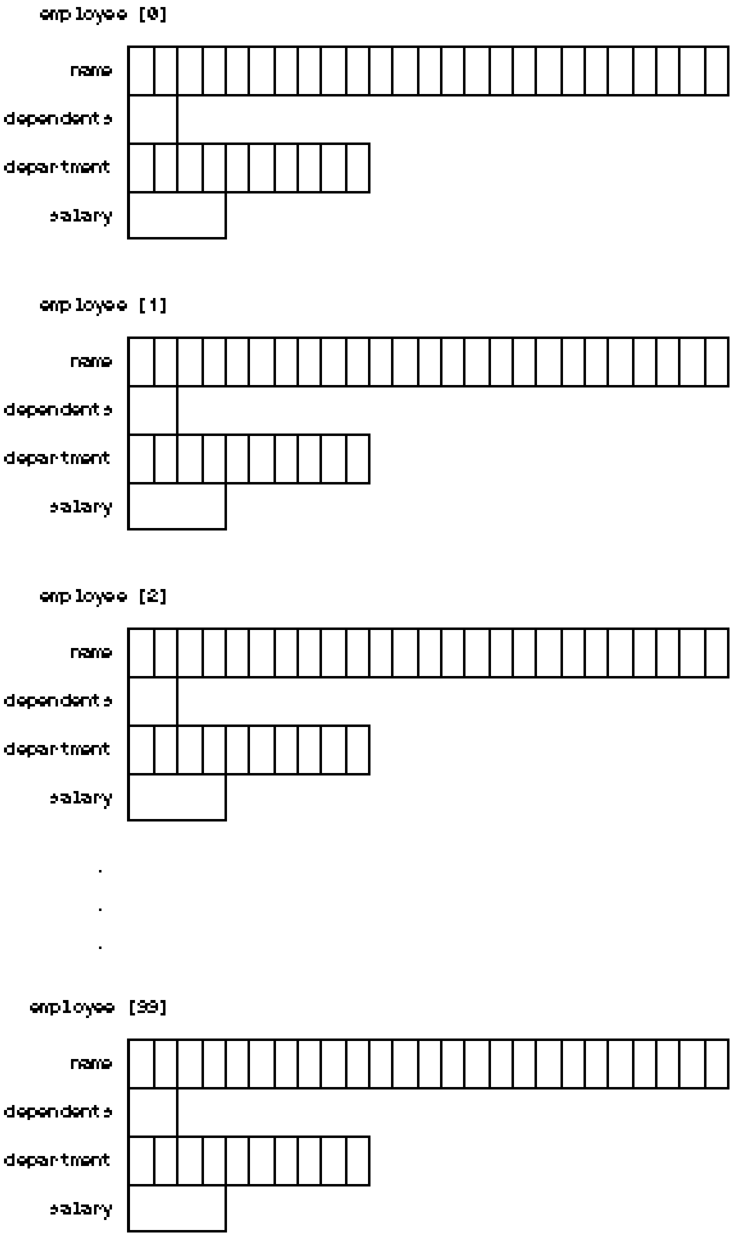


Figure 29.5. The payroll data.

Examples

1. Suppose an employee got married and wanted her name changed in the payroll file. (She happens to be the 45th employee in the array of structures.) Given the payroll structure described in the previous section, this would assign a new name to her structure:

```
strcpy(employee[44].name, “Mary Larson”);   // Assign
                                            // a new name.
```

When you refer to a structure variable using the dot operator, you can use regular commands and functions to process the data in the structure members.

2. A bookstore wants to catalog its inventory of books. The following program creates an array of 100 structures. Each structure contains several types of variables, including arrays. This program is the data-entry portion of a larger inventory system. Study the references to the members to see how member-arrays are used.

```
// Filename: C29BOOK.CPP
// Bookstore data-entry program.
#include <iostream.h>
#include <stdio.h>
#include <ctype.h>

struct inventory
  { char title[25];                    // Book’s title.
    char pub_date[19];                 // Publication date.
    char author[20];                   // Author’s name.
    int num;                           // Number in stock.
    int on_order;                      // Number on order.
    float retail;                      // Retail price.
  };

void main()
{
   inventory book[100];
   int total=0;                // Total books in inventory.
   int ans;
   do     // This program enters data into the structures.
      { cout << “Book #” << (total+1) << “:\n”, (total+1);
     cout << “What is the title? “;
     gets(book[total].title);
     cout << “What is the publication date? “;
     gets(book[total].pub_date);
     cout << “Who is the author? “;
     gets(book[total].author);
     cout << “How many books of this title are there? “;
     cin >> book[total].num;
     cout << “How many are on order? “;
     cin >> book[total].on_order;
     cout << “What is the retail price? “;
     cin >> book[total].retail;
     fflush(stdin);
     cout << “\nAre there more books? (Y/N) “;
     ans=getchar();
     fflush(stdin);              // Discard carriage return.
     ans=toupper(ans);           // Convert to uppercase.
     if (ans==’Y’)
       { total++;
         continue; }
      } while (ans==’Y’);
   return;
}
```

You need much more to make this a usable inventory program. An exercise at the end of this chapter recommends ways you can improve on this program by adding a printing routine and a title and author search. One of the first things you should do is put the data-entry routine in a separate function to make the code more modular. Because this example is so short, and because the program performs only one task (data-entry), there was no advantage to putting the data-entry task in a separate function.

3. Here is a comprehensive example of the steps you might go through to write a C++ program. You should begin to understand the C++ language enough to start writing some advanced programs.

Assume you have been hired by a local bookstore to write a magazine inventory system. You have to track the following:

Magazine title (at most, 25 characters)
Publisher (at most, 20 characters)
Month (1, 2, 3,...12)
Publication year
Number of copies in stock
Number of copies on order
Price of magazine (dollars and cents)

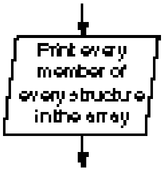
Suppose there is a projected maximum of 1000 magazine titles the store will ever carry. This means you need 1000 occurrences of the structure, not 1000 magazines total. Here is a good structure definition for such an inventory:

```
struct mag_info
   { char title[25];
     char pub[25];
     int month;
     int year;
     int stock_copies;
     int order_copies;
     float price;
   } mags[1000];                 // Define 1000 occurrences.
```

Because this program consists of more than one function, it is best to declare the structure globally, and the structure variables locally in the functions that need them.

This program needs three basic functions: a `main()` controlling function, a data-entry function, and a data printing function. You can add much more, but this is a good start for an inventory system. To keep the length of this example reasonable, assume the user wants to enter several magazines, then print them. (To make the program more "usable," you should add a menu so the user can control when she or he adds and prints the information, and should add more error-checking and editing capabilities.)

Here is an example of the complete data-entry and printing program with prototypes. The arrays of structures are passed between the functions from `main()`.



```
// Filename: C29MAG.CPP
// Magazine inventory program for adding and displaying
// a bookstore’s magazines.
#include <iostream.h>
#include <ctype.h>
#include <stdio.h>

struct mag_info
   { char title[25];
     char pub[25];
     int month;
     int year;
     int stock_copies;
     int order_copies;
     float price;
   };

mag_info fill_mags(struct mag_info mag);
void print_mags(struct mag_info mags[], int mag_ctr);

void main()
{
   mag_info mags[1000];
   int mag_ctr=0;            // Number of magazine titles.
   char ans;

   do
   {                      // Assumes there is
                          // at least one magazine filled.
      mags[mag_ctr] = fill_mags(mags[mag_ctr]);
      cout << “Do you want to enter another magazine? “;
      fflush(stdin);
      ans = getchar();
      fflush(stdin);           // Discards carriage return.
      if (toupper(ans) == ‘Y’)
        { mag_ctr++; }
        } while (toupper(ans) == ‘Y’);
 print_mags(mags, mag_ctr);
```

```
      return;                // Returns to operating system.
}

void print_mags(mag_info mags[], int mag_ctr)
{
   int i;
   for (i=0; i<=mag_ctr; i++)
     { cout << “\n\nMagazine “ << i+1 << “:\n”;// Adjusts for
                                               // subscript.
       cout << “\nTitle: “ << mags[i].title << “\n”;
       cout << “\tPublisher: “ << mags[i].pub << “\n”;
       cout << “\tPub. Month: “ << mags[i].month << “\n”;
       cout << “\tPub. Year: “ << mags[i].year << “\n”;
       cout << “\tIn-stock: “ << mags[i].stock_copies << “\n”;
       cout << “\tOn order: “ << mags[i].order_copies << “\n”;
       cout << “\tPrice: “ << mags[i].price << “\n”;
     }
   return;
}

mag_info fill_mags(mag_info mag)
{
   puts(“\n\nWhat is the title? “);
   gets(mag.title);
   puts(“Who is the publisher? “);
   gets(mag.pub);
   puts(“What is the month (1, 2, ..., 12)? “);
   cin >> mag.month;
   puts(“What is the year? “);
   cin >> mag.year;
   puts(“How many copies in stock? “);
   cin >> mag.stock_copies;
   puts(“How many copies on order? “);
   cin >> mag.order_copies;
   puts(“How much is the magazine? “);
   cin >> mag.price;
   return (mag);
}
```

Review Questions

The answers to the review questions are in Appendix B.

1. True or false: Each element in an array of structures must be the same type.
2. What is the advantage of creating an array of structures rather than using individual variable names for each structure variable?
3. Given the following structure declaration:

```
struct item
  { char part_no[8];
    char descr[20];
    float price;
    int in_stock;
  } inventory[100];
```

   a. How would you assign a price of 12.33 to the 33rd item's in-stock quantity?

   b. How would you assign the first character of the 12th item's part number the value of *X*?

   c. How would you assign the 97th inventory item the same values as the 63rd?

4. Given the following structure declaration:

```
struct item
  { char desc[20];
    int num;
    float cost;
  } inventory[25];
```

   a. What is wrong with the following statement?

```
item[1].cost = 92.32;
```

   b. What is wrong with the following statement?

```
strcpy(inventory.desc, “Widgets”);
```

c. What is wrong with the following statement?

```
inventory.cost[10] = 32.12;
```

Review Exercises

1. Write a program that stores an array of friends' names, phone numbers, and addresses and prints them two ways: with their name, address, and phone number, or with only their name and phone number for a phone listing.
2. Add a sort function to the program in Exercise 1 so you can print your friends' names in alphabetical order. (*Hint:* You have to make the member holding the names a character pointer.)
3. Expand on the book data-entry program, C29BOOK.CPP, by adding features to make it more usable (such as search book by author, by title, and print an inventory of books on order).

Summary

You have mastered structures and arrays of structures. Many useful inventory and tracking programs can be written using structures. By being able to create arrays of structures, you can now create several occurrences of data.

The next step in the process of learning C++ is to save these structures and other data to disk files. The next two chapters explore the concepts of disk file processing.