Structures

Using structures, you have the ability to group data and work with the grouped data as a whole. Business data processing uses the concept of structures in almost every program. Being able to manipulate several variables as a single group makes your programs easier to manage.

This chapter introduces the following concepts:

- Structure definitions
- Initializing structures
- The dot operator (.)
- Structure assignment
- Nested structures

This chapter is one of the last in the book to present new concepts. The remainder of the book builds on the structure concepts you learn in this chapter.

Introduction to Structures

Structures can have members of different data types.

A *structure* is a collection of one or more variable types. As you know, each element in an array must be the same data type, and you must refer to the entire array by its name. Each element (called a *member*) in a structure can be a different data type.

Suppose you wanted to keep track of your CD music collection. You might want to track the following pieces of information about each CD:

Title
Artist
Number of songs
Cost
Date purchased

There would be five members in this CD structure.

TIP: If you have programmed in other computer languages, or if you have ever used a database program, C++ structures are analogous to file records, and members are analogous to fields in those records.

After deciding on the members, you must decide what data type each member is. The title and artist are character arrays, the number of songs is an integer, the cost is floating-point, and the date is another character array. This information is represented like this:

Member Name	*Data Type*
Title	Character array of 25 characters
Artist	Character array of 20 characters
Number of songs	Integer
Cost	Floating-point
Date purchased	Character array of eight characters

Each structure you define can have an associated structure name called a *structure tag.* Structure tags are not required in most cases, but it is generally best to define one for each structure in your program. The structure tag is not a variable name. Unlike array names, which reference the array as variables, a structure tag is simply a label for the structure's format.

You name structure tags yourself, using the same naming rules for variables. If you give the CD structure a structure tag named `cd_collection`, you are informing C++ that the tag called `cd_collection` looks like two character arrays, followed by an integer, a floating-point value, and a final character array.

A structure tag is a label for the structure's format.

A structure tag is actually a newly defined data type that you, the programmer, define. When you want to store an integer, you do not have to define to C++ what an integer is. C++ already recognizes an integer. When you want to store a CD collection's data, however, C++ is not capable of recognizing what format your CD collection takes. You have to tell C++ (using the example being described here) that you need a new data type. That data type will be your structure tag, called `cd_collection` in this example, and it looks like the structure previously described (two character arrays, integer, floating-point, and character array).

NOTE: No memory is reserved for structure tags. A structure tag is your own data type. C++ does not reserve memory for the integer data type until you declare an integer variable. C++ does not reserve memory for a structure until you declare a structure variable.

Figure 28.1 shows the CD structure, graphically representing the data types in the structure. Notice that there are five members and each member is a different data type. The entire structure is called `cd_collection` because that is the structure tag.

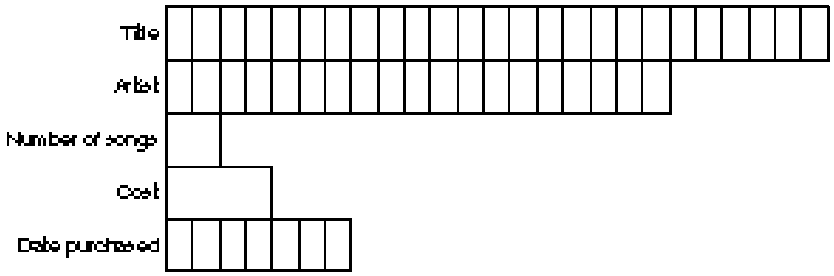
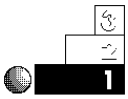


Figure 28.1. The layout of the `cd_collection` structure.

NOTE: The mailing-list application in Appendix F uses a structure to hold people's names, addresses, cities, states, and ZIP codes.

Examples



1. Suppose you were asked to write a program for a company's inventory system. The company had been using a card-file inventory system to track the following items:

   Item name
   Quantity in stock
   Quantity on order
   Retail price
   Wholesale price

   This would be a perfect use for a structure containing five members. Before defining the structure, you have to determine the data types of each member. After asking questions about the range of data (you must know the largest item name, and the highest possible quantity that would appear on order to ensure your data types can hold the data), you decide to use the following structure tag and data types:

Member	*Data Type*
Item name	Character array of 20 characters
Quantity in stock	`long int`
Quantity on order	`long int`
Retail price	`double`
Wholesale price	`double`

2. Suppose the same company also wanted you to write a program to keep track of their monthly and annual salaries and to print a report at the end of the year that showed each month's individual salary and the total salary at the end of the year.

   What would the structure look like? Be careful! This type of data probably does not need a structure. Because all the monthly salaries must be the same data type, a floating-point or a double floating-point array holds the monthly salaries nicely without the complexity of a structure.

   Structures are useful for keeping track of data that must be grouped, such as inventory data, a customer's name and address data, or an employee data file.

Defining Structures

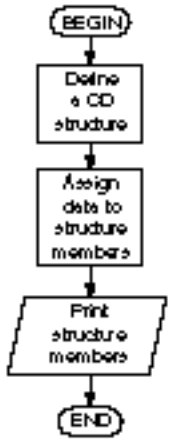
To define a structure, you must use the `struct` statement. The `struct` statement defines a new data type, with more than one member, for your program. The format of the `struct` statement is

```
struct [structure tag]
   {
      member definition;
      member definition;
        :
      member definition;
   } [one or more structure variables];
```

As mentioned earlier, `structure tag` is optional (hence the brackets in the format). Each `member definition` is a normal variable definition, such as `int i;` or `float sales[20];` or any other valid variable definition, including variable pointers if the structure requires a pointer as a member. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables.

If you specify a structure variable, you request C++ to reserve space for that variable. This enables C++ to recognize that the variable is not integer, character, or any other internal data type. C++ also recognizes that the variable must be a type that looks like the structure. It might seem strange that the members do not reserve storage, but they don't. The structure variables do. This becomes clear in the examples that follow.

Here is the way you declare the CD structure:




```
struct cd_collection
   {
     char title[25];
     char artist[20];
     int num_songs;
     float price;
     char date_purch[9];
   } cd1, cd2, cd3;
```

Before going any further, you should be able to answer the following questions about this structure:

- What is the structure tag?
- How many members are there?
- What are the member data types?
- What are the member names?
- How many structure variables are there?
- What are their names?

The structure tag is called `cd_collection`. There are five members, two character arrays, an integer, a floating-point, and a character array. The member names are `title`, `artist`, `num_songs`, `price`, and `date_purch`. There are three structure variables—`cd1`, `cd2`, and `cd3`.

TIP: Often, you can visualize structure variables as a card-file inventory system. Figure 28.2 shows how you might keep your CD collection in a 3-by-5 card file. Each CD takes one card (represented by its structure variable), which contains the information about that CD (the structure members).

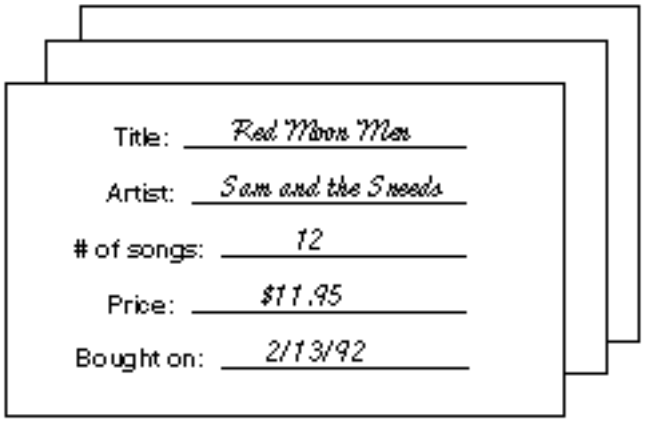
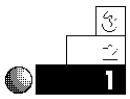


Figure 28.2. Using a card-file CD inventory system.

If you had 1000 CDs, you would have to declare 1000 structure variables. Obviously, you would not want to list that many structure variables at the end of a structure definition. To help define structures for a large number of occurrences, you must define an *array of structures*. Chapter 29, "Arrays of Structures," shows you how to do that. For now, concentrate on familiarizing yourself with structure definitions.

Examples



1. Here is a structure definition of the inventory application described earlier in this chapter.

```
struct inventory
{
   char item_name[20];
   long int in_stock;
   long int order_qty;
   float retail;
   float wholesale;
} item1, item2, item3, item4;
```

Four inventory structure variables are defined. Each structure variable—`item1`, `item2`, `item3`, and `item4`—looks like the structure.

2. Suppose a company wanted to track its customers and personnel. The following two structure definitions would create five structure variables for each structure. This example, having five employees and five customers, is very limited, but it shows how structures can be defined.

```
struct employees
{
   char emp_name[25];              // Employee's full name.
   char address[30];                 // Employee's address.
   char city[10];
   char state[2];
   long int zip;
   double salary;                          // Annual salary.
} emp1, emp2, emp3, emp4, emp5;

struct customers
{
   char cust_name[25];             // Customer's full name.
   char address[30];                 // Customer's address.
   char city[10];
   char state[2];
   long int zip;
   double balance;              // Balance owed to company.
} cust1, cust2, cust3, cust4, cust5;
```

Each structure has similar data. Later in this chapter, you learn how to consolidate similar member definitions by creating nested structures.

TIP: Put comments to the right of members in order to document the purpose of the members.

Initializing Structure Data

You can define a structure's data when you declare the structure.

There are two ways to initialize members of a structure. You can initialize members when you declare a structure, and you can initialize a structure in the body of the program. Most programs lend themselves to the latter method, because you do not always know structure data when you write your program.

Here is an example of a structure declared and initialized at the same time:

```
struct cd_collection
   {
     char title[25];
     char artist[20];
     int num_songs;
     float price;
     char date_purch[9];
   } cd1 = {"Red Moon Men", "Sam and the Sneeds",
            12, 11.95, "02/13/92"};
```

When first learning about structures, you might be tempted to initialize members individually inside the structure, such as

```
char artist[20]="Sam and the Sneeds";       // Invalid
```

You cannot initialize individual members because they are not variables. You can assign only values to variables. The only structure variable in this structure is `cd1`. The braces must enclose the data you initialize in the structure variables, just as they enclose data when you initialize arrays.

This method of initializing structure variables becomes tedious when there are several structure variables (as there usually are). Putting the data in several variables, each set of data enclosed in braces, becomes messy and takes too much space in your code.

More importantly, you usually do not even know the contents of the structure variables. Generally, the user enters data to be stored in structures, or you read them from a disk file.

Use the dot operator to initialize members of structures.

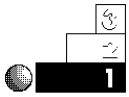
A better approach to initializing structures is to use the *dot operator* (.). The dot operator is one way to initialize individual members of a structure variable in the body of your program. With the dot operator, you can treat each structure member almost as if it were a regular nonstructure variable.

The format of the dot operator is

```
structure_variable_name.member_name
```

A structure variable name must always precede the dot operator, and a member name must always appear after the dot operator. Using the dot operator is easy, as the following examples show.

Examples



1. Here is a simple program using the CD collection structure and the dot operator to initialize the structure. Notice the program treats members as if they were regular variables when combined with the dot operator.

Identify the program and include the necessary header file. Define a CD structure variable with five members. Fill the CD structure variable with data, then print it.

```
// Filename: C28ST1.CPP
// Structure initialization with the CD collection.
#include <iostream.h>
#include <string.h>
void main()
{
   struct cd_collection
   {
     char title[25];
     char artist[20];
     int num_songs;
     float price;
     char date_purch[9];
   } cd1;
```

```
    // Initialize members here.
    strcpy(cd1.title, "Red Moon Men");
    strcpy(cd1.artist, "Sam and the Sneeds");
    cd1.num_songs=12;
    cd1.price=11.95;
    strcpy(cd1.date_purch, "02/13/92");

    // Print the data to the screen.
    cout << "Here is the CD information:\n\n";
    cout << "Title: " << cd1.title << "\n";
    cout << "Artist: " << cd1.artist << "\n";
    cout << "Songs: " << cd1.num_songs << "\n";
    cout << "Price: " << cd1.price << "\n";
    cout << "Date purchased: " << cd1.date_purch << "\n";

    return;
}
```

Here is the output from this program:

```
Here is the CD information:

Title: Red Moon Men
Artist: Sam and the Sneeds
Songs: 12
Price: 11.95
Date purchased: 02/13/92
```

2. By using the dot operator, you can receive structure data from the keyboard with any of the data-input functions you know, such as `cin`, `gets()`, and `get`.

   The following program asks the user for student information. To keep the example reasonably short, only two students are defined in the program.

```
// Filename: C28ST2.CPP
// Structure input with student data.
#include <iostream.h>
#include <string.h>
#include <iomanip.h>
#include <stdio.h>
```

```
void main()
{
   struct students
   {
     char name[25];
     int age;
     float average;
   } student1, student2;

   // Get data for two students.
   cout << "What is first student's name? ";
   gets(student1.name);
   cout << "What is the first student's age? ";
   cin >> student1.age;
   cout << "What is the first student's average? ";
   cin >> student1.average;

   fflush(stdin);    // Clear input buffer for next input.

   cout << "\nWhat is second student's name? ";
   gets(student2.name);
   cout << "What is the second student's age? ";
   cin >> student2.age;
   cout << "What is the second student's average? ";
   cin >> student2.average;

   // Print the data.
   cout << "\n\nHere is the student information you " <<
           "entered:\n\n";
   cout << "Student #1:\n";
   cout << "Name:    " << student1.name << "\n";
   cout << "Age:     " << student1.age << "\n";
   cout << "Average: " << setprecision(2) << student1.average
                       << "\n";

   cout << "\nStudent #2:\n";
   cout << "Name:    " << student2.name << "\n";
   cout << "Age:     " << student2.age << "\n";
   cout << "Average: " << student2.average << "\n";

   return;
}
```

Here is the output from this program:

```
What is first student's name? Larry
What is the first student's age? 14
What is the first student's average? 87.67

What is second student's name? Judy
What is the second student's age? 15
What is the second student's average? 95.38


Here is the student information you entered:

Student #1:
Name:    Larry
Age:     14
Average: 87.67

Student #2:
Name:    Judy
Age:     15
Average: 95.38
```

3. Structure variables are passed by copy, not by address as arrays are. Therefore, if you fill a structure in a function, you must return it to the calling function in order for the calling function to recognize the structure, or use global structure variables, which is generally not recommended.

TIP: A good solution to the local/global structure problem is this: Define your structures globally without any structure variables. Define all your structure variables locally to the functions that need them. As long as your structure definition is global, you can declare local structure variables from that structure. All subsequent examples in this book use this method.

Define structures globally and structure variables locally.

The structure tag plays an important role in the local/global problem. Use the structure tag to define local structure variables. The following program is similar to the previous one. Notice the student structure is defined globally with no

structure variables. In each function, local structure variables are declared by referring to the structure tag. The structure tag keeps you from having to redefine the structure members every time you define a new structure variable.

```
// Filename: C28ST3.CPP
// Structure input with student data passed to functions.
#include <iostream.h>
#include <string.h>
#include <stdio.h>
#include <iomanip.h>
struct students fill_structs(struct students student_var);
void pr_students(struct students student_var);

struct students                      // A global structure.
   {
     char name[25];
     int age;
     float average;
   };                             // No memory reserved.

void main()
{
   students student1, student2;      // Defines two
                                     // local variables.

   // Call function to fill structure variables.
   student1 = fill_structs(student1);         // student1
                 // is passed by copy, so it must be
                 // returned for main() to recognize it.
   student2 = fill_structs(student2);

   // Print the data.
   cout << "\n\nHere is the student information you";
   cout << " entered:\n\n";
   pr_students(student1);   // Prints first student's data.
   pr_students(student2);  // Prints second student's data.

   return;
}
```

```
struct students fill_structs(struct students student_var)
{
   // Get student's data
   fflush(stdin);    // Clears input buffer for next input.
   cout << "What is student's name? ";
   gets(student_var.name);
   cout << "What is the student's age? ";
   cin >> student_var.age;
   cout << "What is the student's average? ";
   cin >> student_var.average;

   return (student_var);
}

void pr_students(struct students student_var)
{
   cout << "Name:    " << student_var.name << "\n";
   cout << "Age:     " << student_var.age << "\n";
   cout << "Average: " << setprecision(2) <<
                          student_var.average << "\n";
   return;
}
```

The prototype and definition of the `fill_structs()` function might seem complicated, but it follows the same pattern you have seen throughout this book. Before a function name, you must declare `void` or put the return data type if the function returns a value. `fill_structs()` does return a value, and the type of value it returns is `struct students`.

4. Because structure data is nothing more than regular variables grouped together, feel free to calculate using structure members. As long as you use the dot operator, you can treat structure members just as you would other variables.

   The following example asks for a customer's balance and uses a discount rate, included in the customer's structure, to calculate a new balance. To keep the example short, the structure's data is initialized at variable declaration time.

   This program does not actually require structures because only one customer is used. Individual variables could have

been used, but they don't illustrate the concept of calculating with structures.

```
// Filename: C28CUST.CPP
// Updates a customer balance in a structure.
#include <iostream.h>
#include <iomanip.h>

struct customer_rec
   {
      char cust_name[25];
      double balance;
      float dis_rate;
   } ;

void main()
{
   struct customer_rec customer = {"Steve Thompson",
                                   431.23, .25};

   cout << "Before the update, " << customer.cust_name;
   cout << " has a balance of $" << setprecision(2) <<
           customer.balance << "\n";

   // Update the balance
   customer.balance *= (1.0-customer.dis_rate);

   cout << "After the update, " << customer.cust_name;
   cout << " has a balance of $" << customer.balance << "\n";
   return;
}
```

5. You can copy the members of one structure variable to the members of another as long as both structures have the same format. Some older versions of C++ require you to copy each member individually when you want to copy one structure variable to another, but AT&T C++ makes duplicating structure variables easy.

EXAMPLE

Being able to copy one structure variable to another will seem more meaningful when you read Chapter 29, “Arrays of Structures.”

The following program declares three structure variables, but initializes only the first one with data. The other two are then initialized by assigning the first structure variable to them.


```
// Filename: C28STCPY.CPP
// Demonstrates assigning one structure to another.
#include <iostream.h>
#include <iomanip.h>

struct student
{
   char st_name[25];
   char grade;
   int age;
   float average;
};

void main()
{
   student std1 = {“Joe Brown”, ‘A’, 13, 91.4};
   struct student std2, std3;          // Not initialized

   std2 = std1;             // Copies each member of std1
   std3 = std1;             // to std2 and std3.


   cout << “The contents of std2:\n”;
   cout << std2.st_name << “ “ << std2.grade << “ “;
   cout << std2.age << “ “ << setprecision(1) << std2.average
        << “\n\n”;

   cout << “The contents of std3:\n”;
   cout << std3.st_name << “ “ << std3.grade << “ “;
   cout << std3.age << “ “ << std3.average << “\n”;
   return;
}
```

Here is the output from the program:

```
The contents of std2
Joe Brown, A, 13, 91.4

The contents of std3
Joe Brown, A, 13, 91.4
```

Notice each member of `std1` was assigned to `std2` and `std3` with two single assignments.

Nested Structures

C++ gives you the ability to nest one structure definition in another. This saves time when you are writing programs that use similar structures. You have to define the common members only once in their own structure and then use that structure as a member in another structure.

The following two structure definitions illustrate this point:

```
struct employees
{
   char emp_name[25];          // Employee's full name.
   char address[30];           // Employee's address.
   char city[10];
   char state[2];
   long int zip;
   double salary;              // Annual salary.
};

struct customers
{
   char cust_name[25];         // Customer's full name.
   char address[30];           // Customer's address.
   char city[10];
   char state[2];
   long int zip;
   double balance;             // Balance owed to company.
};
```

These structures hold different data. One structure is for employee data and the other holds customer data. Even though the data should be kept separate (you don't want to send a customer a paycheck!), the structure definitions have much overlap and can be consolidated by creating a third structure.

Suppose you created the following structure:

```
struct address_info
{
   char address[30];        // Common address information.
   char city[10];
   char state[2];
   long int zip;
};
```

This structure could then be used as a member in the other structures like this:

```
struct employees
{
   char emp_name[25];         // Employee's full name.
   address_info e_address;    // Employee's address.
   double salary;             // Annual salary.
};

struct customers
{
   char cust_name[25];        // Customer's full name.
   address_info c_address;    // Customer's address.
   double balance;            // Balance owed to company.
};
```

It is important to realize there are a total of three structures, and that they have the tags `address_info`, `employees`, and `customers`. How many members does the `employees` structure have? If you answered three, you are correct. There are three members in both `employees` and `customers`. The `employees` structure has the structure of a character array, followed by the `address_info` structure, followed by the double floating-point member, `salary`.

Figure 28.3 shows how these structures look.

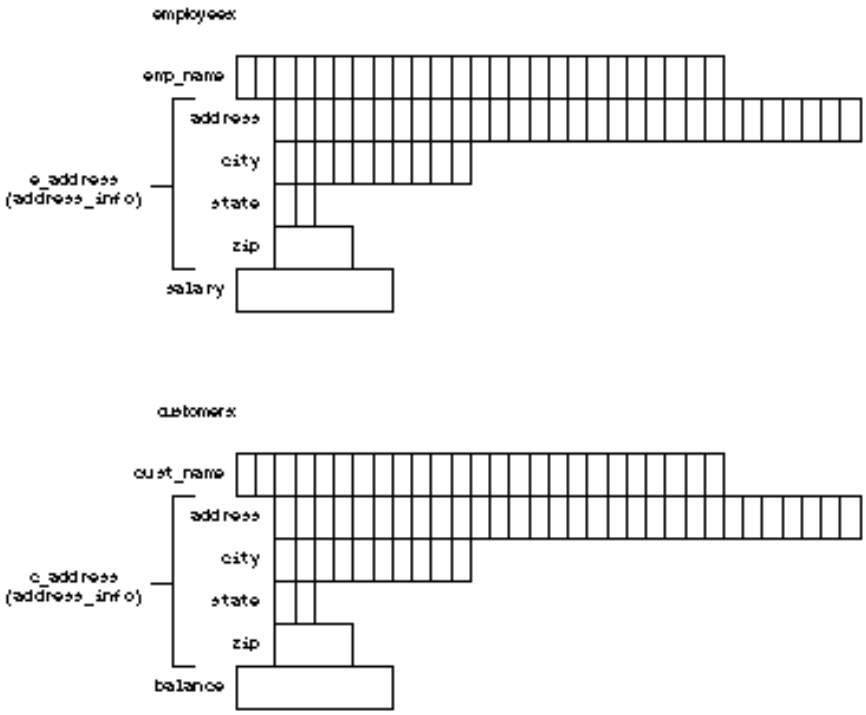


Figure 28.3. Defining a nested structure.

When you define a structure, that structure becomes a new data type in the program and can be used anywhere a data type (such as `int`, `float`, and so on) can appear.

You can assign members values using the dot operator. To assign the customer balance a number, type something like this:

```
customer.balance = 5643.24;
```

The nested structure might seem to pose a problem. How can you assign a value to one of the nested members? By using the dot operator, you must nest the dot operator just as you nest the structure definitions. You would assign a value to the customer's ZIP code like this:

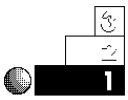
```
customer.c_address.zip = 34312;
```

To assign a value to the employee's ZIP code, you would do this:

```
employee.e_address.zip = 59823;
```

Review Questions

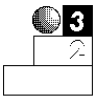
The answers to the review questions are in Appendix B.



1. What is the difference between structures and arrays?
2. What are the individual elements of a structure called?
3. What are the two ways to initialize members of a structure?

4. Do you pass structures by copy or by address?
5. True or false: The following structure definition reserves storage in memory:

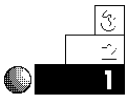
```
struct crec
   { char name[25];
     int age;
     float sales[5];
     long int num;
   }
```



6. Should you declare a structure globally or locally?
7. Should you declare a structure variable globally or locally?
8. How many members does the following structure declaration contain?

```
struct item
   {
     int quantity;
     part_rec item_desc;
     float price;
     char date_purch[8];
   };
```

Review Exercises



1. Write a structure in a program that tracks a video store's tape inventory. Be sure the structure includes the tape title, the length of the tape (in minutes), the initial purchase price of the tape, the rental price of the tape, and the date of the movie's release.

2. Write a program using the structure you declared in Exercise 1. Define three structure variables and initialize them when you declare the variables with data. Print the data to the screen.

3. Write a teacher's program to keep track of 10 students' names, ages, letter grades, and IQs. Use 10 different structure variable names and retrieve the data for the students in a `for` loop from the keyboard. Print the data on the printer when the teacher finishes entering the information for all the students.

Summary

With structures, you have the ability to group data in more flexible ways than with arrays. Your structures can contain members of different data types. You can initialize the structures either at declaration time or during the program with the dot operator.

Structures become even more powerful when you declare arrays of structure variables. Chapter 29, "Arrays of Structures," shows you how to declare several structure variables without giving them each a different name. This enables you to step through structures much quicker with loop constructs.