Structures

Assist. Prof. Dr.
Ahmed Hashim Mohammed

CHAPTER

4

IN THIS CHAPTER

- Structures
- Enumerations

We've seen variables of simple data types, such as `float`, `char`, and `int`. Variables of such types represent one item of information: a height, an amount, a count, and so on. But just as groceries are organized into bags, employees into departments, and words into sentences, it's often convenient to organize simple variables into more complex entities. The C++ construction called the structure is one way to do this.

The first part of this chapter is devoted to structures. In the second part we'll look at a related topic: enumerations.

Structures

A structure is a collection of simple variables. The variables in a structure can be of different types: Some can be `int`, some can be `float`, and so on. (This is unlike the array, which we'll meet later, in which all the variables must be the same type.) The data items in a structure are called the members of the structure.

In books on C programming, structures are often considered an advanced feature and are introduced toward the end of the book. However, for C++ programmers, structures are one of the two important building blocks in the understanding of objects and classes. In fact, the syntax of a structure is almost identical to that of a class. A structure (as typically used) is a collection of data, while a class is a collection of both data and functions. So by learning about structures we'll be paving the way for an understanding of classes and objects. Structures in C++ (and C) serve a similar purpose to records in some other languages such as Pascal.

A Simple Structure

Let's start off with a structure that contains three variables: two integers and a floating-point number. This structure represents an item in a widget company's parts inventory. The structure is a kind of blueprint specifying what information is necessary for a single part. The company makes several kinds of widgets, so the widget model number is the first member of the structure. The number of the part itself is the next member, and the final member is the part's cost. (Those of you who consider part numbers unexciting need to open your eyes to the romance of commerce.)

The program PARTS defines the structure `part`, defines a structure variable of that type called `part1`, assigns values to its members, and then displays these values.

```
// parts.cpp
// uses parts inventory to demonstrate structures
#include <iostream>
using namespace std;
```

```
////////////////////////////////////////////////////////////////
struct part                    //declare a structure
   {
   int modelnumber;            //ID number of widget
   int partnumber;             //ID number of widget part
   float cost;                 //cost of part
   };
////////////////////////////////////////////////////////////////
int main()
   {
   part part1;                 //define a structure variable

   part1.modelnumber = 6244; //give values to structure members
   part1.partnumber = 373;
   part1.cost = 217.55F;
                               //display structure members
   cout << "Model "     << part1.modelnumber;
   cout << ", part "    << part1.partnumber;
   cout << ", costs $" << part1.cost << endl;
   return 0;
   }
```

The program's output looks like this:

```
Model 6244, part 373, costs $217.55
```

The PARTS program has three main aspects: defining the structure, defining a structure variable, and accessing the members of the structure. Let's look at each of these.

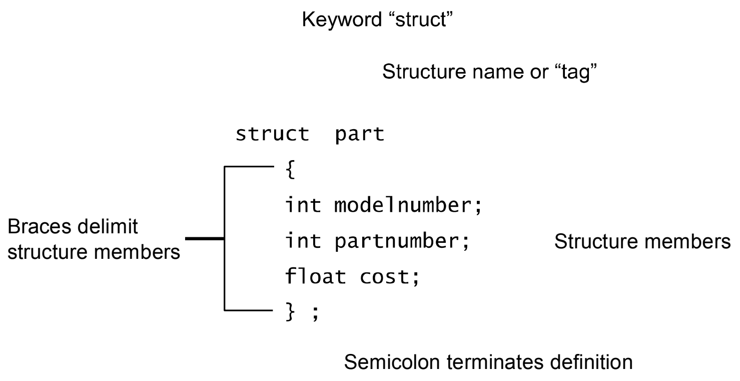
Defining the Structure

The structure definition tells how the structure is organized: It specifies what members the structure will have. Here it is:

```
struct part
   {
   int modelnumber;
   int partnumber;
   float cost;
   };
```

Syntax of the Structure Definition

The keyword `struct` introduces the structure definition. Next comes the structure name or tag, which is `part`. The declarations of the structure members—`modelnumber`, `partnumber`, and `cost`—are enclosed in braces. A semicolon follows the closing brace, terminating the entire structure. Note that this use of the semicolon for structures is unlike the usage for a block of code. As we've seen, blocks of code, which are used in loops, decisions, and functions, are also delimited by braces. However, they don't use a semicolon following the final brace. Figure 4.1 shows the syntax of the structure declaration.


Keyword "struct"
Structure name or "tag"
Braces delimit structure members
Structure members
Semicolon terminates definition

```
struct  part
    {
    int modelnumber;
    int partnumber;
    float cost;
    } ;
```

FIGURE 4.1
Syntax of the structure definition.

Use of the Structure Definition

The structure definition definition serves only as a blueprint for the creation of variables of type `part`. It does not itself create any structure variables; that is, it does not set aside any space in memory or even name any variables. This is unlike the definition of a simple variable, which does set aside memory. A structure definition is merely a specification for how structure variables will look when they are defined. This is shown in Figure 4.2.

It's not accidental that this description sounds like the distinction we noted between classes and objects in Chapter 1, "The Big Picture." As we'll see, an object has the same relationship to its class that a variable of a structure type has to the structure definition.

Defining a Structure Variable

The first statement in `main()`

```
part part1;
```

defines a variable, called `part1`, of type structure `part`. This definition reserves space in memory for `part1`. How much space? Enough to hold all the members of `part1`—namely `modelnumber`, `partnumber`, and `cost`. In this case there will be 4 bytes for each of the two `int`s (assuming a 32-bit system), and 4 bytes for the `float`. Figure 4.3 shows how `part1` looks in memory. (The figure shows 2-byte integers.)

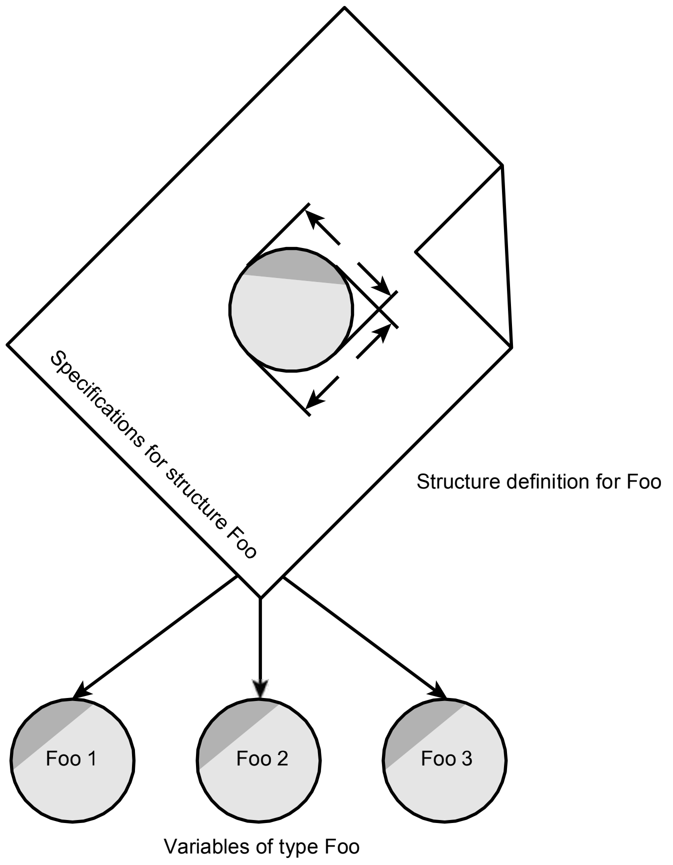

Specifications for structure Foo
Structure definition for Foo
Foo 1
Foo 2
Foo 3
Variables of type Foo

FIGURE 4.2
Structures and structure variables.

In some ways we can think of the `part` structure as the specification for a new data type. This will become more clear as we go along, but notice that the format for defining a structure variable is the same as that for defining a basic built-in data type such as `int`:

```
part part1;
int var1;
```

This similarity is not accidental. One of the aims of C++ is to make the syntax and the operation of user-defined data types as similar as possible to that of built-in data types. (In C you need to include the keyword `struct` in structure definitions, as in `struct part part1;`. In C++ the keyword is not necessary.)

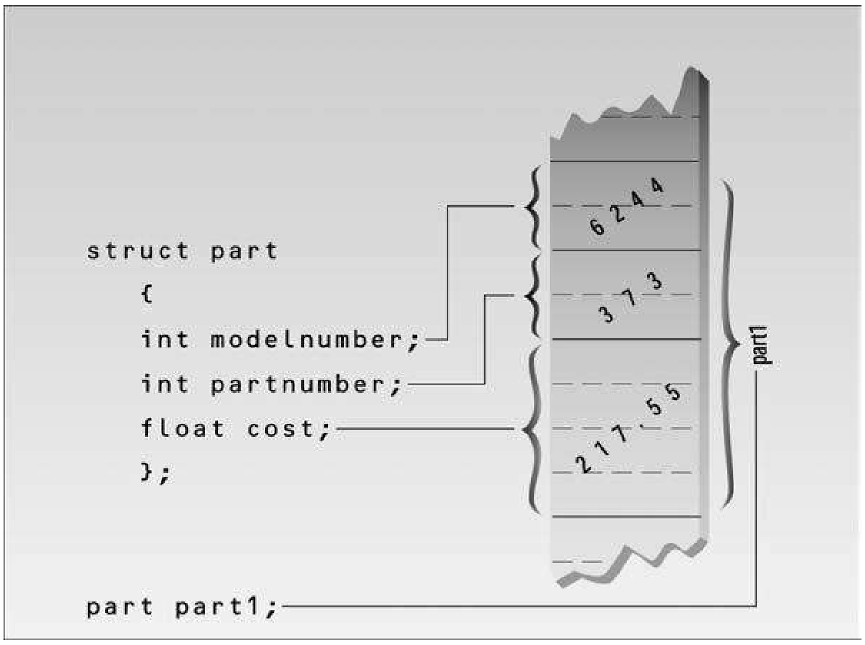


FIGURE 4.3
Structure members in memory.

Accessing Structure Members

Once a structure variable has been defined, its members can be accessed using something called the dot operator. Here's how the first member is given a value:

```
part1.modelnumber = 6244;
```

The structure member is written in three parts: the name of the structure variable (`part1`); the dot operator, which consists of a period (`.`); and the member name (`modelnumber`). This means "the `modelnumber` member of `part1`." The real name of the dot operator is member access operator, but of course no one wants to use such a lengthy term.

Remember that the first component of an expression involving the dot operator is the name of the specific structure variable (`part1` in this case), not the name of the structure definition (`part`). The variable name must be used to distinguish one variable from another, such as `part1`, `part2`, and so on, as shown in Figure 4.4.

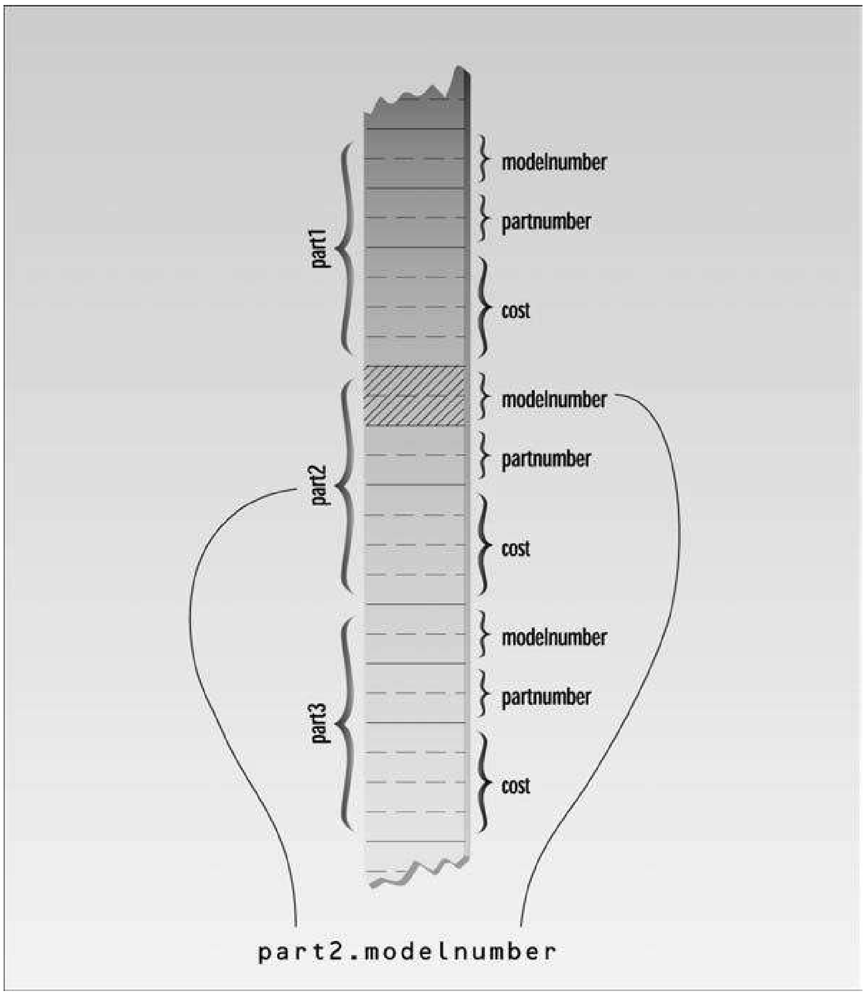


FIGURE 4.4
The dot operator.

Structure members are treated just like other variables. In the statement `part1.modelnumber = 6244;`, the member is given the value 6244 using a normal assignment operator. The program also shows members used in `cout` statements such as

```
cout << "\nModel " << part1.modelnumber;
```

These statements output the values of the structure members.

Other Structure Features

Structures are surprisingly versatile. Let's look at some additional features of structure syntax and usage.

Initializing Structure Members

The next example shows how structure members can be initialized when the structure variable is defined. It also demonstrates that you can have more than one variable of a given structure type (we hope you suspected this all along).

Here's the listing for PARTINIT:

```
// partinit.cpp
// shows initialization of structure variables
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
struct part                    //specify a structure
   {
   int modelnumber;            //ID number of widget
   int partnumber;             //ID number of widget part
   float cost;                 //cost of part
   };
////////////////////////////////////////////////////////////////
int main()
   {                           //initialize variable
   part part1 = { 6244, 373, 217.55F };
   part part2;                 //define variable
                               //display first variable
   cout << "Model "    << part1.modelnumber;
   cout << ", part "   << part1.partnumber;
   cout << ", costs $" << part1.cost << endl;

   part2 = part1;              //assign first variable to second
                               //display second variable
   cout << "Model "    << part2.modelnumber;
   cout << ", part "   << part2.partnumber;
   cout << ", costs $" << part2.cost << endl;
   return 0;
   }
```

This program defines two variables of type `part`: `part1` and `part2`. It initializes `part1`, prints out the values of its members, assigns `part1` to `part2`, and prints out its members.

Here's the output:

```
Model 6244, part 373, costs $217.55
Model 6244, part 373, costs $217.55
```

Not surprisingly, the same output is repeated since one variable is made equal to the other.

The `part1` structure variable's members are initialized when the variable is defined:

```
part part1 = { 6244, 373, 217.55 };
```

The values to be assigned to the structure members are surrounded by braces and separated by commas. The first value in the list is assigned to the first member, the second to the second member, and so on.

Structure Variables in Assignment Statements

As can be seen in PARTINIT, one structure variable can be assigned to another:

```
part2 = part1;
```

The value of each member of `part1` is assigned to the corresponding member of `part2`. Since a large structure can have dozens of members, such an assignment statement can require the computer to do a considerable amount of work.

Note that one structure variable can be assigned to another only when they are of the same structure type. If you try to assign a variable of one structure type to a variable of another type, the compiler will complain.

A Measurement Example

Let’s see how a structure can be used to group a different kind of information. If you’ve ever looked at an architectural drawing, you know that (at least in the United States) distances are measured in feet and inches. (As you probably know, there are 12 inches in a foot.) The length of a living room, for example, might be given as 15’–8”, meaning 15 feet plus 8 inches. The hyphen isn’t a negative sign; it merely separates the feet from the inches. This is part of the English system of measurement. (We’ll make no judgment here on the merits of English versus metric.) Figure 4.5 shows typical length measurements in the English system.

Suppose you want to create a drawing or architectural program that uses the English system. It will be convenient to store distances as two numbers, representing feet and inches. The next example, ENGLSTRC, gives an idea of how this could be done using a structure. This program will show how two measurements of type `Distance` can be added together.

```
// englstrc.cpp
// demonstrates structures using English measurements
#include <iostream>
using namespace std;
/////////////////////////////////////////////////////////////
struct Distance                //English distance
   {
   int feet;
   float inches;
   };
////////////////////////////////////////////////////////////////
```