Lab Session 09

Structures in C++ Language.

Objectives:

1. Illustration of structures.
2. To learn the syntax and semantics of the Structure in C++ programming language.
3. Demonstrate a thorough understanding of structures by designing and implementing programs

Structures:

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.

Associating multiple values together

Now that you can store a single value in an array, it's possible for you to write programs that deal with a lot of data. As you work with more data, there will be times that you have several pieces of data that are all associated together. For example, you might want to store screen coordinates (x and y values) for players in a videogame along with their names. Right now, you could do that with three separate arrays:

```
int x_coordinates[ 10 ];
int y_coordinates[ 10 ];
string names[ 10 ];
```

But you'd have to remember that each array is matched up with another, so if you moved the position of an element in one of the arrays, you would have to move the associated element in the other two arrays.

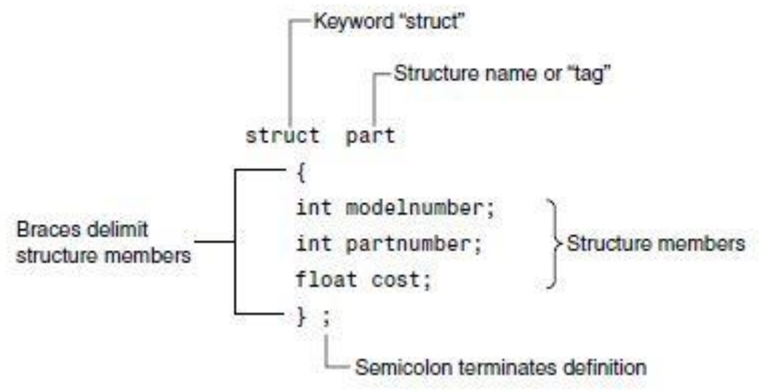
This would also get pretty unwieldy when you need to keep track of a fourth value. You'd have add another array and keep it in sync with the original three. Fortunately, the people who design programming languages are not masochists, so there is a better way of associating values together: structures. Structures allow you to store different values in variables under the same variable name. Structures are useful whenever several pieces of data need to be grouped together.

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 1 shows the syntax of the structure declaration.


```
struct  part
{
int modelnumber;
int partnumber;
float cost;
} ;
```
Keyword "struct"
Structure name or "tag"
Braces delimit structure members
Structure members
Semicolon terminates definition

Syntax of Structure

The format for defining a structure is

```
struct SpaceShip
{
      int x_coordinate;
      int y_coordinate;
      string name;
}; // <- Notice that pesky semicolon; you must include it
```

Here, SpaceShip is the name of the particular type of structure that we are defining. In other words, you have created your own type, just like double or int, which you can use to declare a variable, like so:

```
SpaceShip my_ship;
```

The names x_coordinate, y_coordinate and name are the fields of our new type. Wait, fields, what does that mean exactly?

Here's the story: we've just created a compound type—a variable that stores multiple values that are all associated with each other (like two screen coordinates, or a first and last name). The way you tell the variable which one of those values you want is by naming the field you want to access. It's like having two separate variables with different names, except that the two variables are grouped together and you have a consistent way of naming them. You can think of a structure as a form (think driver's license application) with fields—the form stores a lot of data, and each field of the form is a particular piece of that related data. Declaring a structure is the way to define the form, and declaring a variable of that structure's type creates a copy of that form that you can fill out and use to store a bunch of data.

To access fields, you put in the name of the variable (not the name of the structure—each variable has its own separate values for its fields) followed by a dot '.', followed by the name of the field:

```
// declare the variable
SpaceShip my_ship;
// use it
my_ship.x_coordinate = 40;
my_ship.y_coordinate = 40;
my_ship.name = "USS Enterprise (NCC-1701-D)";
```

As you can see, you can have many fields in a structure, practically as many as you want, and they do not all have to be the same type.

Let's now look at an example program that demonstrates reading in the names of five players in a game (game not included), which will combine arrays and structures:

```
#include <iostream>
using namespace std;
struct PlayerInfo
{
int skill_level;
string name;
};
using namespace std;
int main ()
{
      // like normal variable types, you can make arrays of structures
      PlayerInfo players[ 5 ];
      for ( int i = 0; i < 5; i++ )
      {
            cout << "Please enter the name for player : " << i << '\n';
            // first access the element of the array, using normal
            // array syntax; then access the field of the structure
```

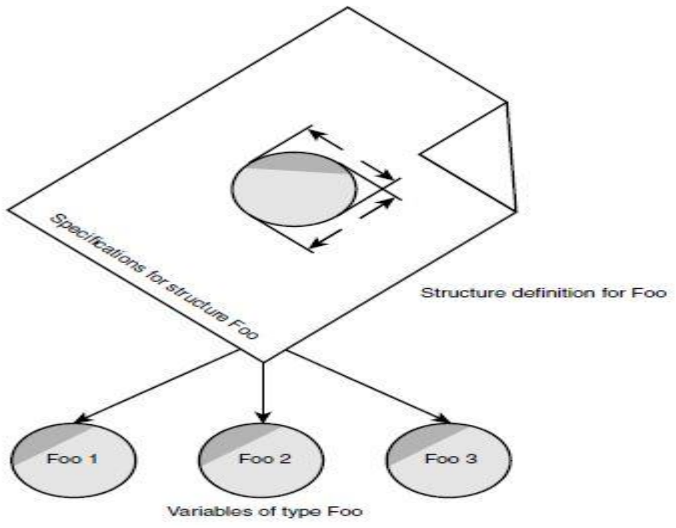
```
        // using the '.' syntax
        cin >> players[ i ].name;
        cout << "Please enter the skill level for " << players[ i
        ].name<< \n';
        cin >> players[ i ].skill_level;
    }
    for ( int i = 0; i < 5; ++i )
    {
        cout << players[ i ].name << " is at skill level " << players[ i
        ].skill_level << '\n';
    }
}
```

The struct PlayerInfo declares that it has two fields: the name of a player, and the player's skill_level. Since you can use PlayerInfo like any other variable type (e.g. int), you can create an array of players. When you create an array of structures, you treat each element of the array like you would treat a single structure instance—to access a field of the first structure in the array, you would just write players[0].name to access to the name of the player in the first element of the array.

This program takes advantage of the ability to combine arrays and structures to read in the information for five different players, with two different pieces of data, in a single for loop, and then display that information in a second loop. There's no need to have multiple related arrays for each individual piece of data. You don't need separate player_names and player_skill_level arrays.

Use of the Structure Definition

The structure definitiondefinition 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 2.


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

Structure and Structure Variables

Defining a Structure Variable

The first statement in main()

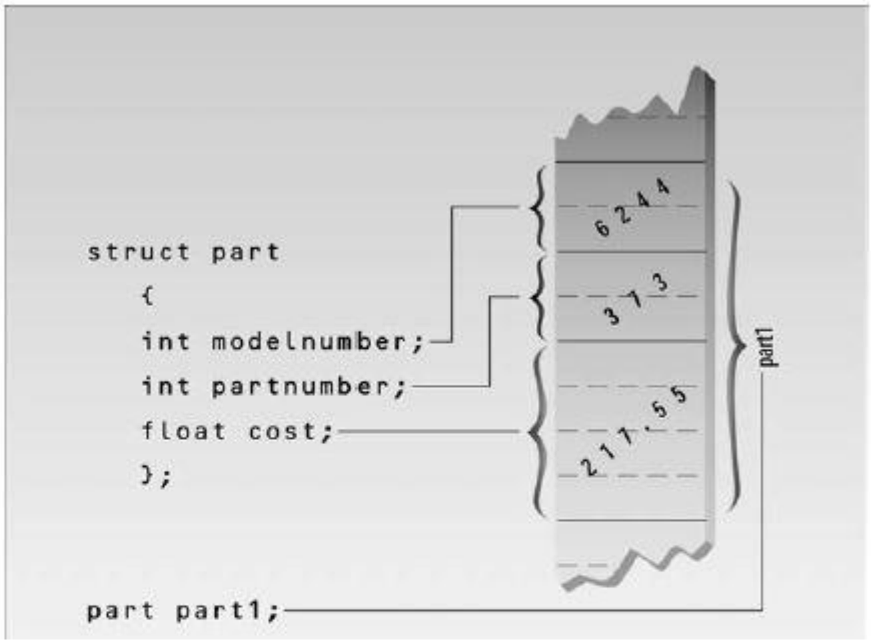
part part1;

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 intsm (assuming a 32-bit system), and 4 bytes for the float. Figure 3 shows how part1 looks in memory. (The figure shows 2-byte integers.)

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.)


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

part part1;
```

Structure members in memory

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.

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;
```

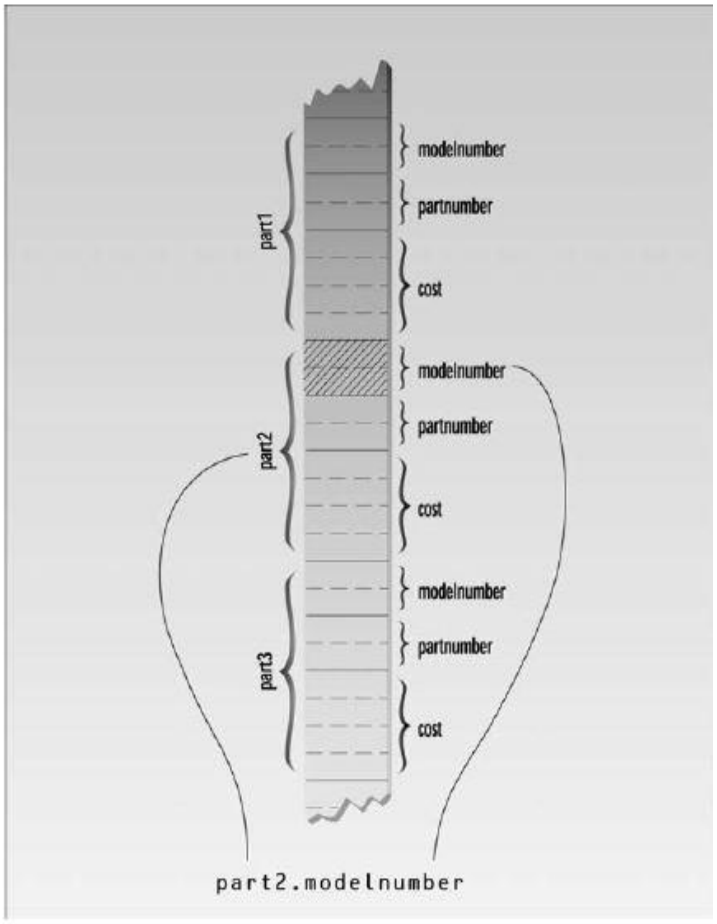
The structure member is written in three parts: the name of the structure variable (part1); the dot operator, which consists of a period (.); and the member name (modelnumber). This means "the modelnumber member of part1." The real name of the dot operator is *member access operator*, but of course no 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.

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 Features of Structure:

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:

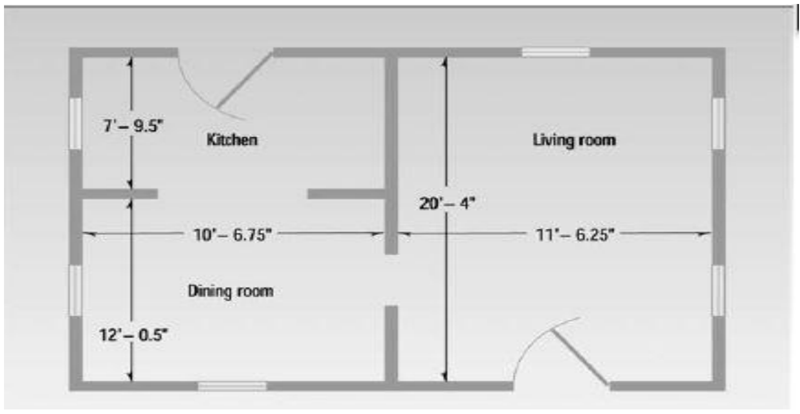
```
part2 = part1;
```

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.

Example 01:

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 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;
        };
int main()
        {
        Distance d1, d3; //define two lengths
        Distance d2 = { 11, 6.25 }; //define & initialize one length get length d1 from user
        cout << "\nEnter feet: "; cin >> d1.feet;
        cout << "Enter inches: "; cin >> d1.inches;
        //add lengths d1 and d2 to get d3
        d3.inches = d1.inches + d2.inches; //add the inches
        d3.feet = 0; //(for possible carry)
        if(d3.inches >= 12.0) //if total exceeds 12.0, then decrease inches by 12.0
                {
                        d3.inches -= 12.0; //and
                        d3.feet++; //increase feet by 1
                }
        d3.feet += d1.feet + d2.feet; //add the feet
        //displays all lengths
        cout << d1.feet << “\’-” << d1.inches << “\” + “;
        cout << d2.feet << “\’-” << d2.inches << “\” = “;
        cout << d3.feet << “\’-” << d3.inches << “\”\n”;
        return 0;
}
```

Here the structure Distance has two members: feet and inches. The inches variable may have a fractional part, so we’ll use type float for it. Feet are always integers, so we’ll use type int for them.

We define two such distances, d1 and d3, without initializing them, while we initialize another, d2, to 11'–6.25". The program asks the user to enter a distance in feet and inches, and assigns this distance to d1. (The inches value should be smaller than 12.0.) It then adds the distance d1 to d2, obtaining the total distance d3. Finally the program displays the two initial distances and the newly calculated total distance. Here’s some output:

```
Enter feet: 10
Enter inches: 6.75
10’-6.75” + 11’-6.25” = 22’-1”
```

Notice that we can’t add the two distances with a program statement like

```
d3 = d1 + d2; // can’t do this in ENGLSTRC
```

Why not? Because there is no routine built into C++ that knows how to add variables of type Distance. The + operator works with built-in types like float, but not with types we define ourselves, like Distance. (However, one of the benefits of using classes, as we’ll see in Chapter 8, “Operator Overloading,” is the ability to add and perform other operations on userdefined data types.)

Structures within Structures:

You can nest structures within other structures. Here’s a variation on the ENGLSTRC program that shows how this looks. In this program we want to create a data structure that stores the dimensions of a typical room: its length and width. Since we’re working with English distances, we’ll use two variables of type Distance as the length and width variables.

```
struct Room
        {
        Distance length;
        Distance width;
        }
```

Example 02:
Here’s a program, ENGLAREA, that uses the Room structure to represent a room.

```
#include <iostream>
using namespace std;

struct Distance         //English distance
        {
        int feet;
        float inches;
        };
struct Room //rectangular area
        {
        Distance length; //length of rectangle
        Distance width; //width of rectangle
        };
int main()
        {
        Room dining; //define a room
        dining.length.feet = 13; //assign values to room
        dining.length.inches = 6.5;
        dining.width.feet = 10;
        dining.width.inches = 0.0;
        //convert length & width
        float l = dining.length.feet + dining.length.inches/12;
        float w = dining.width.feet + dining.width.inches/12;
        //find area and display it
        cout << “Dining room area is “ << l * w
        << “ square feet\n” ;
        return 0;
        }
```

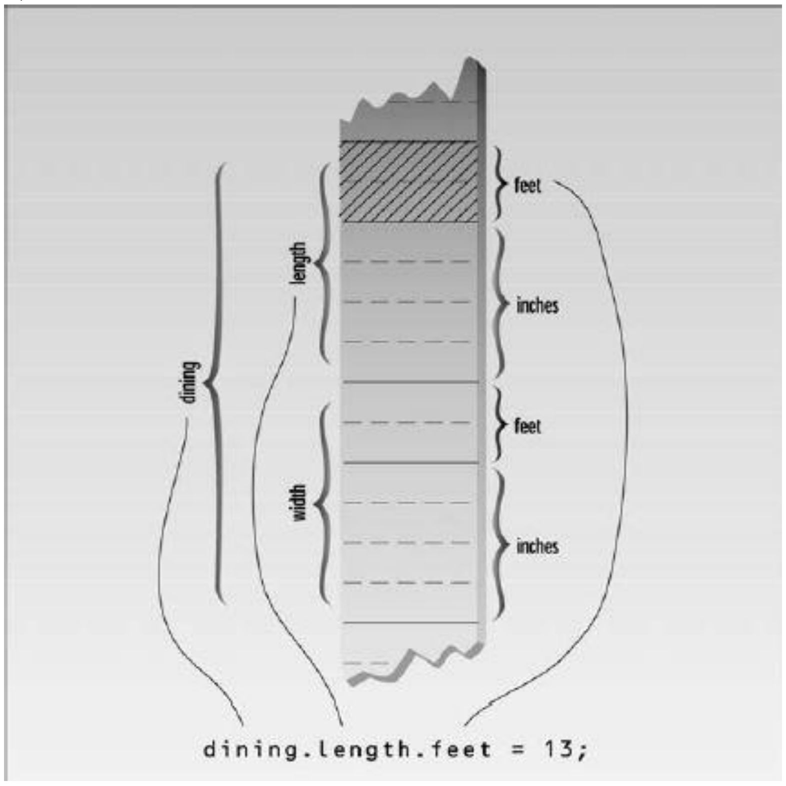
This program defines a single variable—dining—of type Room, in the line
Room dining; // variable dining of type Room
It then assigns values to the various members of this structure.

Accessing Nested Structure Members

Because one structure is nested inside another, we must apply the dot operator twice to access the structure members.

dining.length.feet = 13;

In this statement, dining is the name of the structure variable, as before; length is the name of a member in the outer structure (Room); and feet is the name of a member of the inner structure (Distance). The statement means "take the feet member of the length member of the variable dining and assign it the value 13." Figure 6 shows how this works.

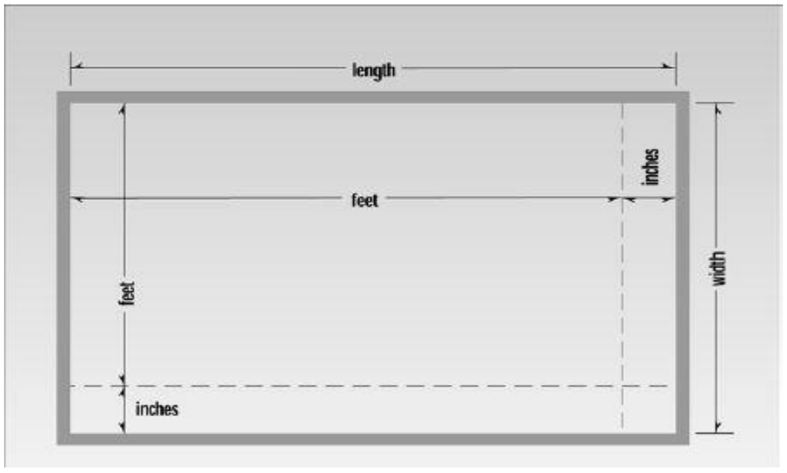


dot Operator and nested structure

Once values have been assigned to members of dining, the program calculates the floor area of the room, as shown in Figure 7.

To find the area, the program converts the length and width from variables of type Distance to variables of type float, l, and w, representing distances in feet. The values of l and w are found by adding the feet member of Distance to the inches member divided by 12.

The feet member is converted to type float automatically before the addition is performed, and the result is type float. The l and w variables are then multiplied together to obtain the area.



Area in feet and inches

User-Defined Type Conversions

Note that the program converts two distances of type Distance to two distances of type float: the variables l and w. In effect it also converts the room's area, which is stored as a structure of type Room (which is defined as two structures of type Distance), to a single floating-point number representing the area in square feet. Here's the output:

```
Dining room area is 135.416672 square feet
```

Converting a value of one type to a value of another is an important aspect of programs that employ user-defined data types.

Initializing Nested Structures

How do you initialize a structure variable that itself contains structures? The following statement initializes the variable dining to the same values it is given in the ENGLAREA program:

```
Room dining = { {13, 6.5}, {10, 0.0} };
```

Each structure of type Distance, which is embedded in Room, is initialized separately. Remember that this involves surrounding the values with braces and separating them with commas. The first Distance is initialized to

```
{13, 6.5}
```

and the second to

```
{10, 0.0}
```

These two Distance values are then used to initialize the Room variable; again, they are surrounded with braces and separated by commas.

Depth of Nesting

In theory, structures can be nested to any depth. In a program that designs apartment buildings, you might find yourself with statements like this one:

```
apartment1.laundry_room.washing_machine.width.feet
```

Tasks/Assignment:

1. Create a database of five students using structures in which following information of a student is stored.
    a. Name of the Student
    b. Registration #
    c. Department
    d. Program
    e. Semester
    f. Email
    g. Contact #
2. Modify Example 02 using structures then get the information from user of an apartment dining room and two bed rooms the total area of dining room and two bed rooms shouldn’t exceed 1200 sq. ft.