Lecture 2: Structures

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. 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.
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 1 shows the syntax of the structure declaration.

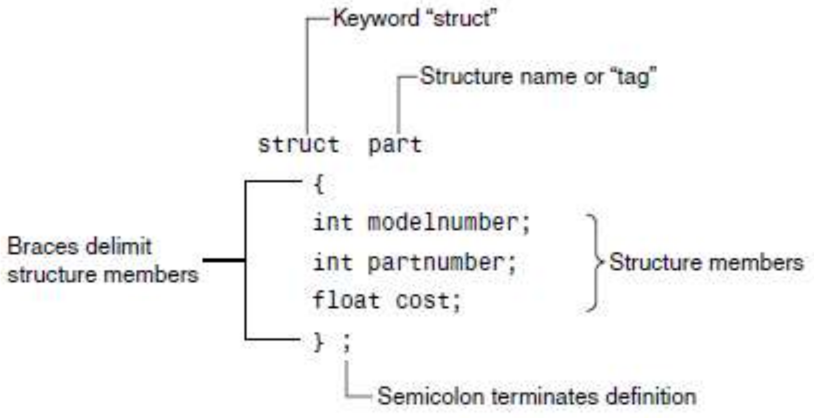


FIGURE 1

Syntax of the structure definition.

Use of the Structure Definition

The structure 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 2.

As we 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 ints (assuming a 32-bit system), and 4 bytes for the float. Figure 3 shows how part1 looks in memory.

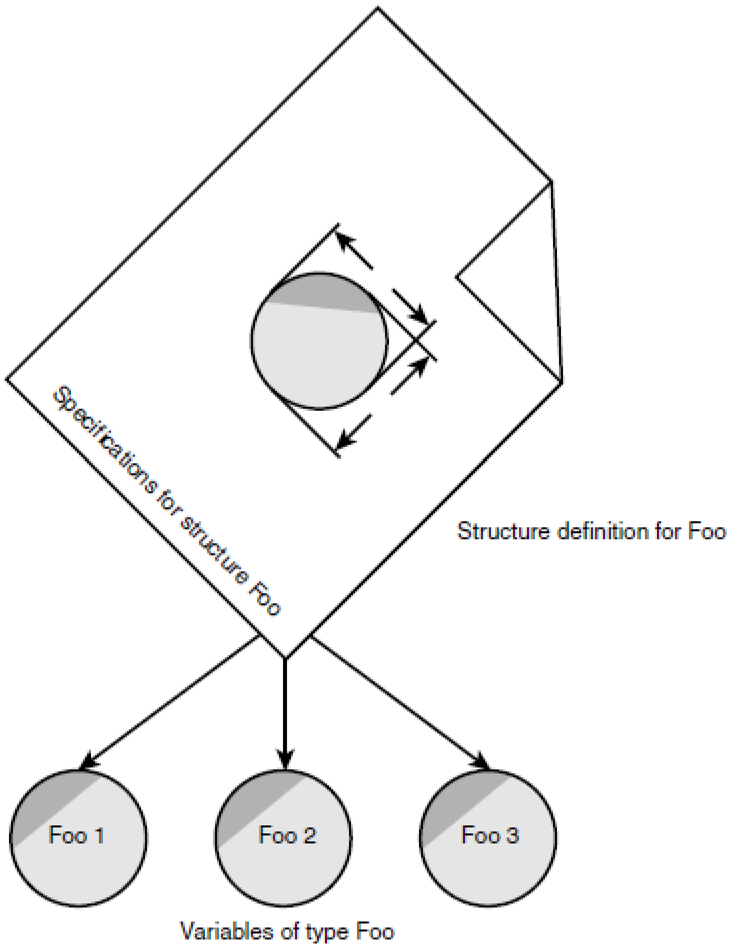

Specifications for structure Foo
Structure definition for Foo
Foo 1
Foo 2
Foo 3
Variables of type Foo

FIGURE 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;
```

Note: (In C you need to include the keyword `struct` in structure definitions, as in struct part part1;. In C++ the keyword is not necessary.)

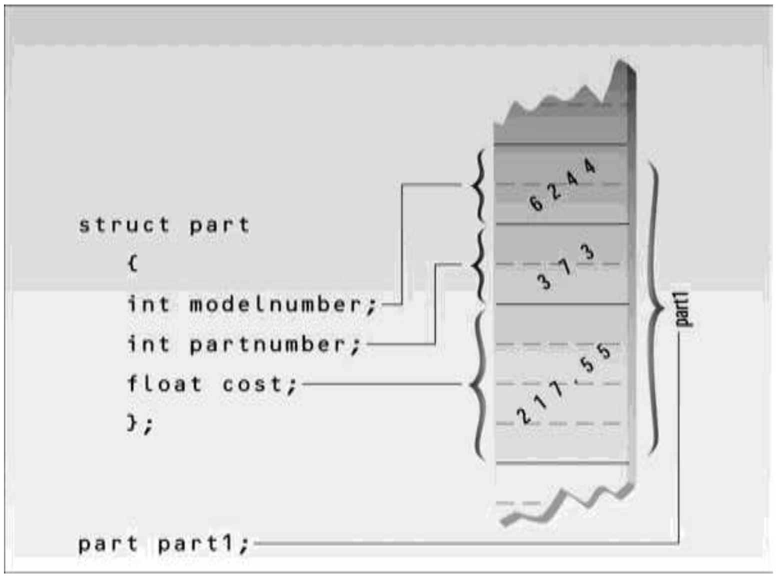


FIGURE 3
Structure members in memory.

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*.
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.

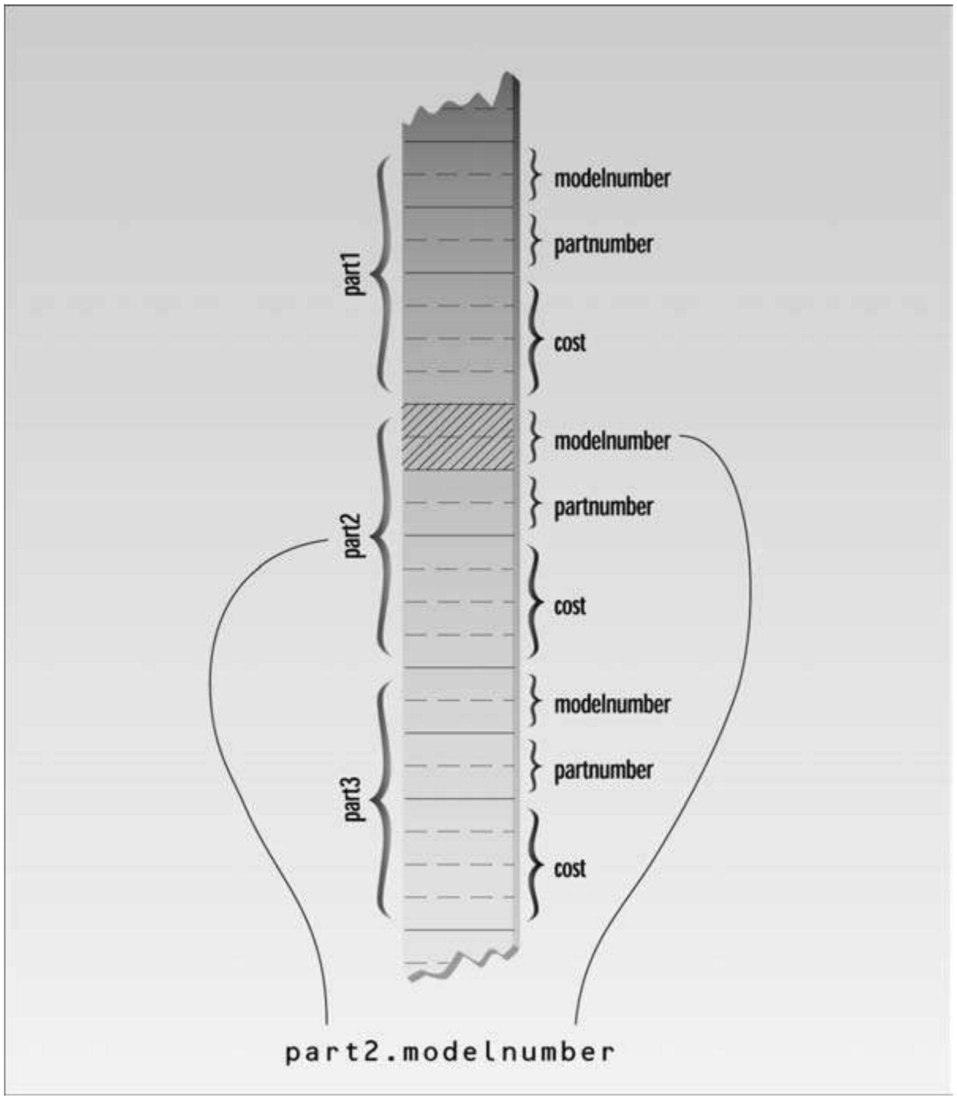


FIGURE 4
The dot operator.

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.

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. 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

                                //display all lengths
   cout << d1.feet << "\'-" << d1.inches << "\" + ";
   cout << d2.feet << "\'-" << d2.inches << "\" = ";
   cout << d3.feet << "\'-" << d3.inches << "\"\n";
   return 0;
   }
```

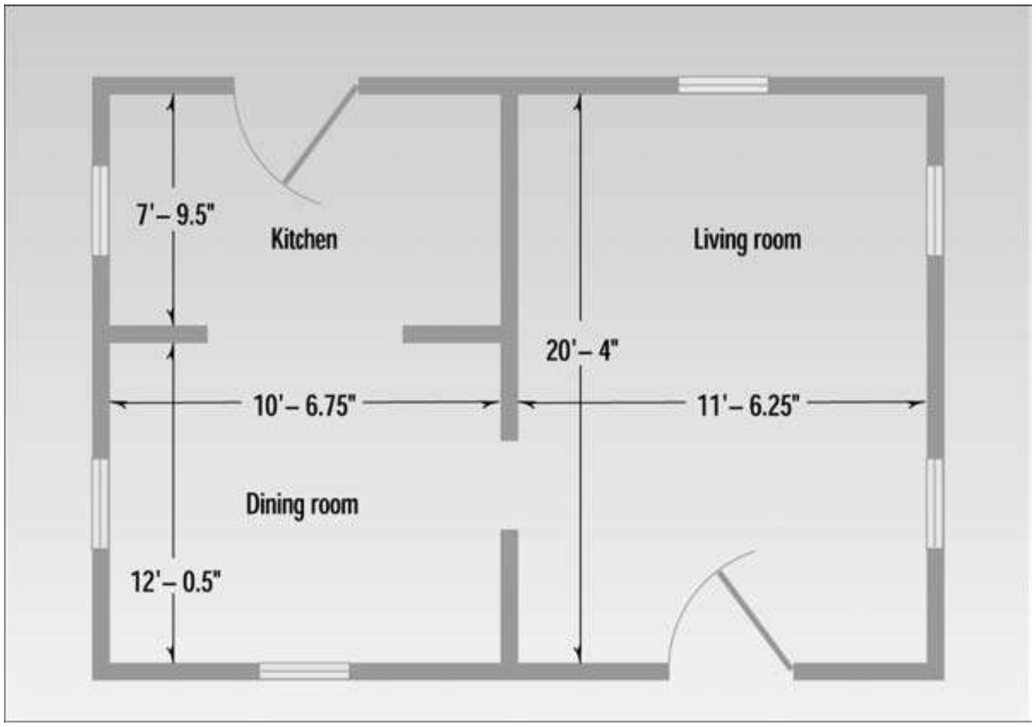


FIGURE 5
Measurements in the English system.

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”
```

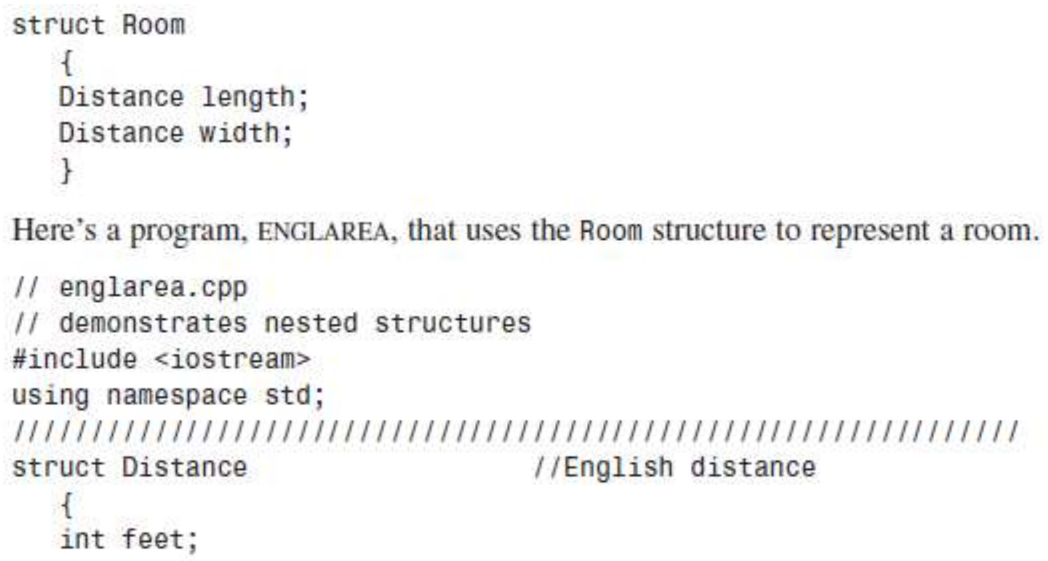
Notice that we can't add the two distances with a program statement like

```
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 lecture, "Operator Overloading," is the ability to add and perform other operations on user defined 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;
   }
```

Here's a program, ENGLAREA, that uses the `Room` structure to represent a room.

```
// englarea.cpp
// demonstrates nested structures
#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.

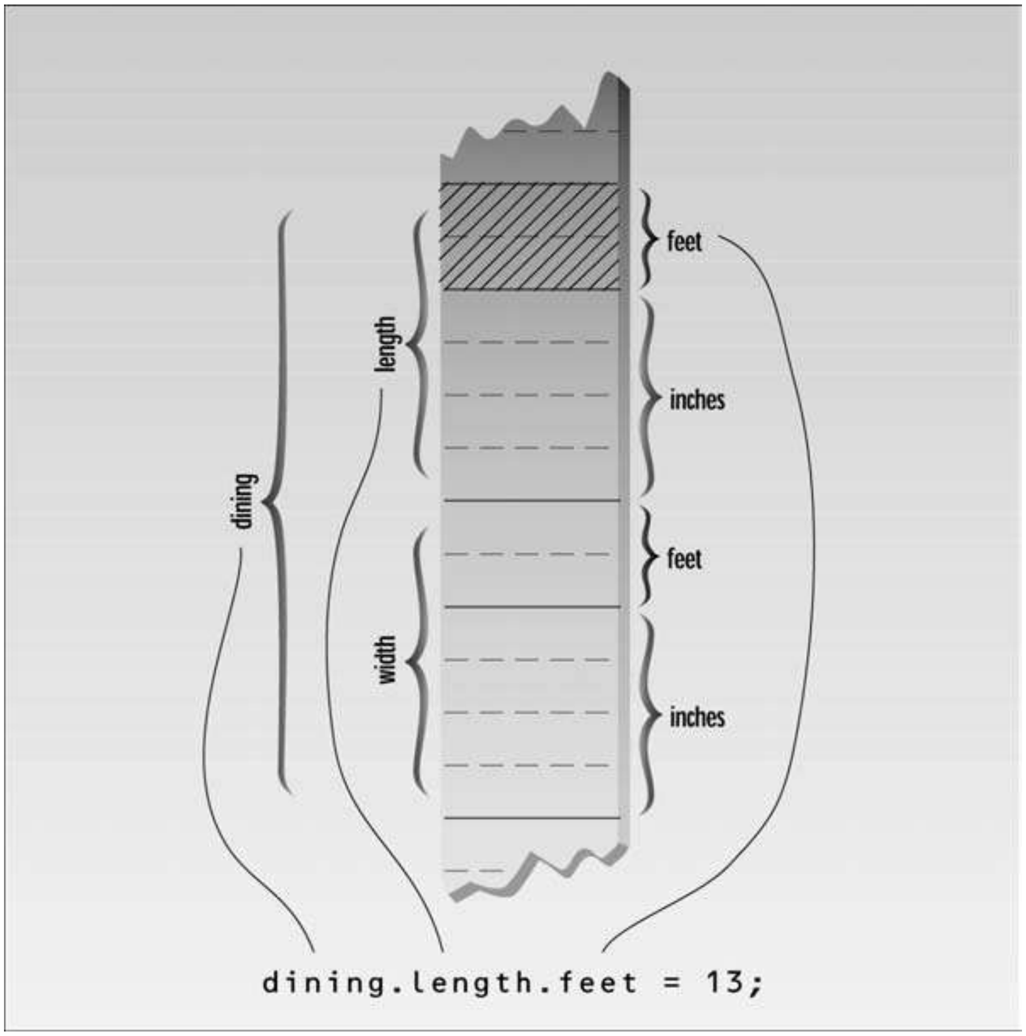


FIGURE 6
Dot operator and nested structures.

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.

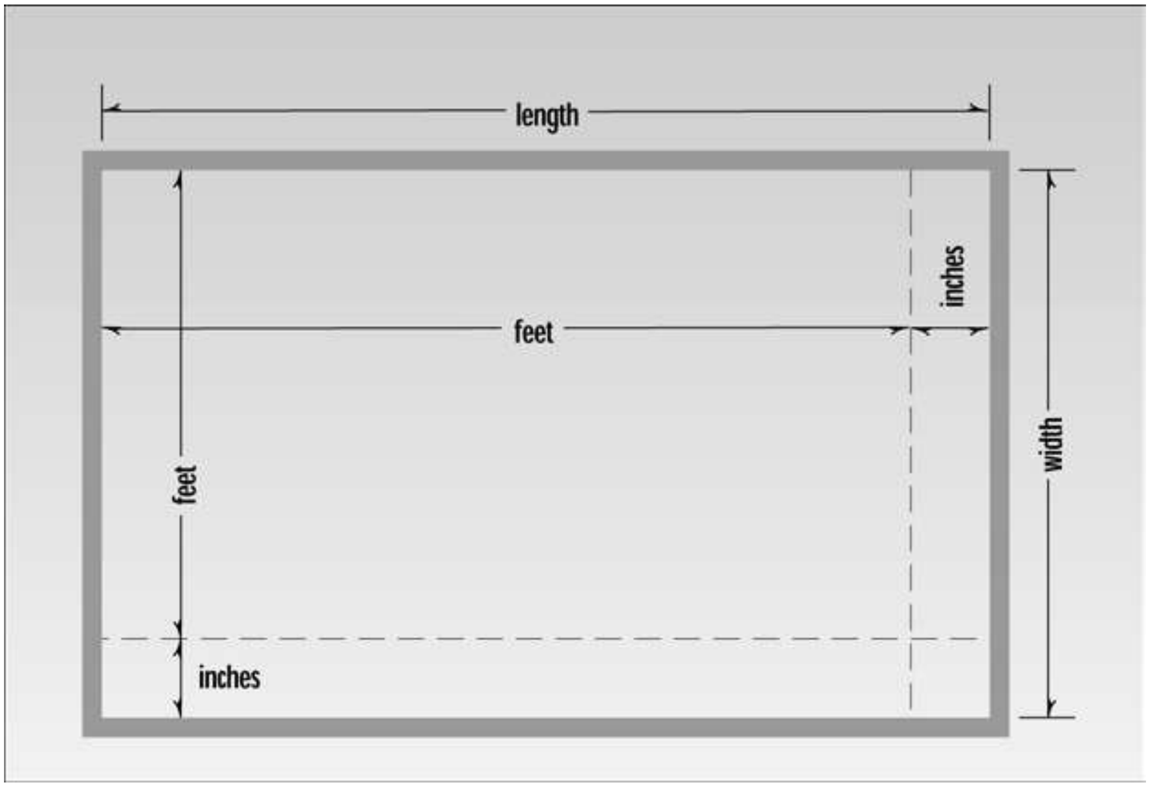


FIGURE 7
Area in feet and inches.

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
```

Structures and Classes

We must confess to having misled you slightly on the capabilities of structures. It’s true that structures are usually used to hold data only, and classes are used to hold both data and functions.
However, in C++, structures can in fact hold both data and functions. (In C they can hold only data.) The syntactical distinction between structures and classes in C++ is minimal, so they can in theory be used almost interchangeably. But most C++ programmers use structures as we have in this chapter, exclusively for data. Classes are usually used to hold both data and functions, as we’ll see in lecture, “Objects and Classes.”

Enumerations

As we've seen, structures can be looked at as a way to provide user-defined data types. A different approach to defining your own data type is the *enumeration*. This feature of C++ is less crucial than structures. You can write perfectly good object-oriented programs in C++ without knowing anything about enumerations. However, they are very much in the spirit of C++, in that, by allowing you to define your own data types, they can simplify and clarify your programming.

Days of the Week

Enumerated types work when you know in advance a finite (usually short) list of values that a data type can take on. Here's an example program, DAYENUM, that uses an enumeration for the days of the week:

```
// dayenum.cpp
// demonstrates enum types
#include <iostream>
using namespace std;
                              //specify enum type
enum days_of_week { Sun, Mon, Tue, Wed, Thu, Fri, Sat };

int main()
   {
   days_of_week day1, day2;   //define variables
                              //of type days_of_week

 day1 = Mon;                //give values to
 day2 = Thu;                //variables

 int diff = day2 - day1;    //can do integer arithmetic
 cout << "Days between = " << diff << endl;

 if(day1 < day2)            //can do comparisons
    cout << "day1 comes before day2\n";
 return 0;
 }
```

An enum declaration defines the set of all names that will be permissible values of the type.

These permissible values are called *enumerators*. The enum type days_of_week has seven enumerators: Sun, Mon, Tue, and so on, up to Sat. Figure 8 shows the syntax of an enum declaration.

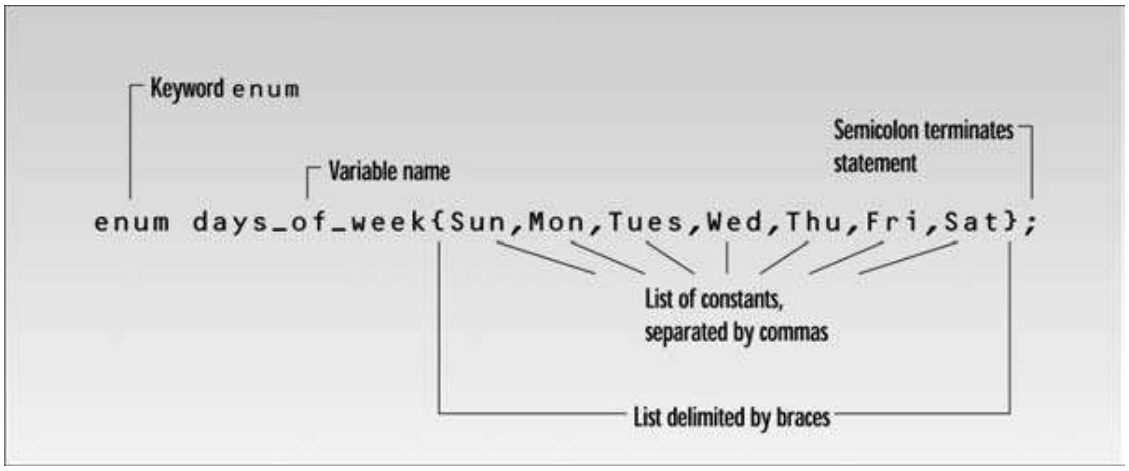

Keyword enum
Variable name
Semicolon terminates statement
enum days_of_week{Sun,Mon,Tues,Wed,Thu,Fri,Sat};
List of constants, separated by commas
List delimited by braces

FIGURE 8
Syntax of enum *specifier.*

An *enumeration* is a list of all possible values. This is unlike the specification of an int, for example, which is given in terms of a range of values. In an enum you must give a specific name to every possible value. Figure 9 shows the difference between an int and an enum.
Once you've declared the enum type days_of_week as shown, you can define variables of this type. DAYENUM has two such variables, day1 and day2, defined in the statement

```
days_of_week day1, day2;
```

(In C you must use the keyword enum before the type name, as in enum

```
days_of_week day1, day2;
```

In C++ this isn't necessary.)

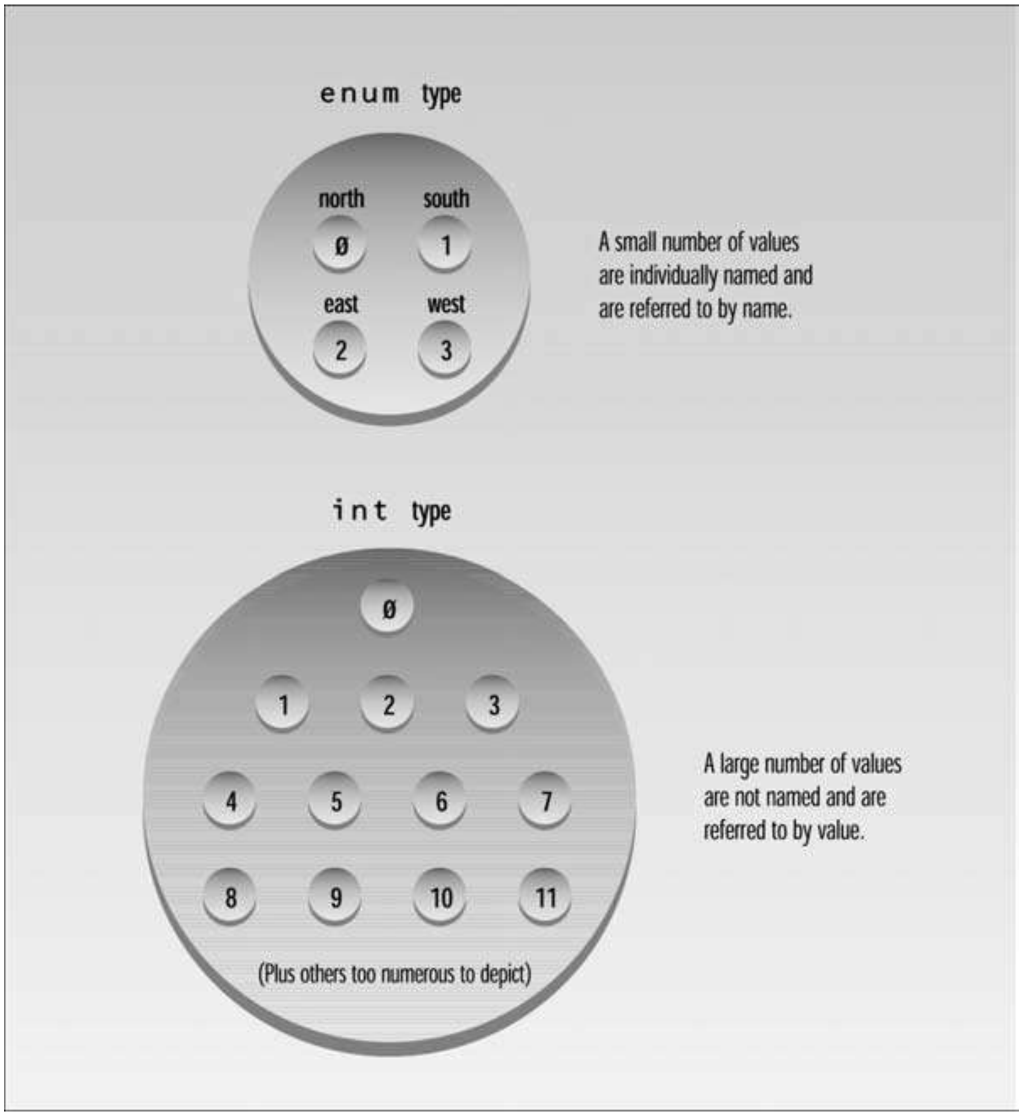

enum type
north
ø
south
1
east
2
west
3
A small number of values are individually named and are referred to by name.
int type
ø
1
2
3
4
5
6
7
8
9
10
11
(Plus others too numerous to depict)
A large number of values are not named and are referred to by value.

FIGURE 9
Usage of ints *and* enums.

Variables of an enumerated type, like day1 and day2, can be given any of the values listed in the enum declaration. In the example we give them the values Mon and Thu. You can’t use values that weren’t listed in the declaration. Such statements as

```
day1 = halloween;
```

are illegal.

You can use the standard arithmetic operators on enum types. In the program we subtract two values. You can also use the comparison operators, as we show. Here's the program's output:

```
Days between = 3
day1 comes before day2
```

The use of arithmetic and relational operators doesn't make much sense with some enum types. For example, if you have the declaration

```
enum pets { cat, dog, hamster, canary, ocelot };
```

then it may not be clear what expressions like `dog + canary` or `(cat < hamster)` mean. Enumerations are treated internally as integers. This explains why you can perform arithmetic and relational operations on them. Ordinarily the first name in the list is given the value 0, the next name is given the value 1, and so on. In the DAYENUM example, the values Sun through Sat are stored as the integer values 0–6.
Arithmetic operations on enum types take place on the integer values. However, although the compiler knows that your enum variables are really integers, you must be careful of trying to take advantage of this fact. If you say

```
day1 = 5;
```

the compiler will issue a warning (although it will compile). It's better to forget—whenever possible—that enums are really integers.

One Thing or Another

Our next example counts the words in a phrase typed in by the user. Unlike the earlier CHCOUNT example, however, it doesn't simply count spaces to determine the number of words. Instead it counts the places where a string of nonspace characters changes to a space, as shown in Figure 10.

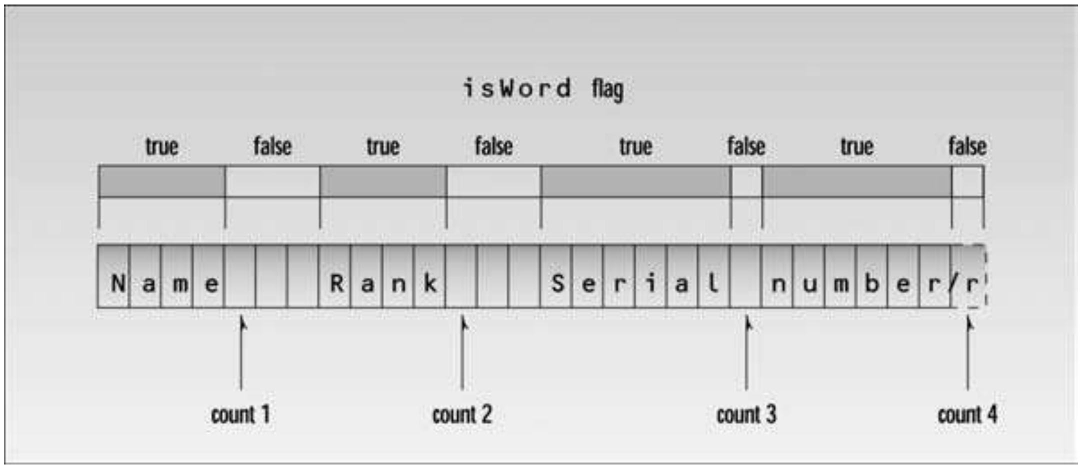


FIGURE 10
Operation of the WDCOUNT program.

This way you don’t get a false count if you type multiple spaces between words. (It still doesn’t handle tabs and other whitespace characters.) Here’s the listing for WDCOUNT: This example shows an enumeration with only two enumerators.

```
// wdcount.cpp
// demonstrates enums, counts words in phrase
#include <iostream>
using namespace std;
#include <conio.h>              //for getche()

enum itsaWord { NO, YES };      //NO=0, YES=1

int main()
   {
   itsaWord isWord = NO;        //YES when in a word,
                                //NO when in whitespace
   char ch = 'a';               //character read from keyboard
   int wordcount = 0;           //number of words read

   cout << "Enter a phrase:\n";
   do {
      ch = getche();            //get character
      if(ch==' ' || ch=='\r')   //if white space,
         {
         if( isWord == YES )    //and doing a word,
            {                   //then it's end of word
            wordcount++;        //count the word
            isWord = NO;        //reset flag
            }
         }                      //otherwise, it's
      else                      //normal character
         if( isWord == NO )     //if start of word,
            isWord = YES;       //then set flag
      } while( ch != '\r' );    //quit on Enter key
   cout << "\n---Word count is " << wordcount << "---\n";
   return 0;
   }
```

The program cycles in a do loop, reading characters from the keyboard. It passes over (nonspace) characters until it finds a space. At this point it counts a word. Then it passes over spaces until it finds a character, and again counts characters until it finds a space. Doing this requires the program to remember whether it's in the middle of a word, or in the middle of a string of spaces. It remembers this with the enum variable isWord. This variable is defined to be of type itsaWord. This type is specified in the statement

```
enum itsaWord { NO, YES };
```

Variables of type itsaWord have only two possible values: NO and YES. Notice that the list starts with NO, so this value will be given the value 0—the value that indicates false. (We could also use a variable of type bool for this purpose.)

The `isword` variable is set to NO when the program starts. When the program encounters the first nonspace character, it sets `isWord` to YES to indicate that it's in the middle of a word. It keeps this value until the next space is found, at which point it's set back to NO. Behind the scenes, NO has the value 0 and YES has the value 1, but we avoid making use of this fact. We could have used `if(isWord)` instead of `if(isWord == YES)`, and `if(!isWord)` instead of `if(isWord == NO)`, but this is not good style.
Note also that we need an extra set of braces around the second if statement in the program, so that the else will match the first if. Another approach to a yes/no situation such as that in WDCOUNT is to use a variable of type bool. This may be a little more straightforward, depending on the situation.

Other Examples
Here are some other examples of enumerated data declarations, to give you a feeling for possible uses of this feature:

```
enum months { Jan, Feb, Mar, Apr, May, Jun,
              Jul, Aug, Sep, Oct, Nov, Dec };

enum switch { off, on };

enum meridian { am, pm };

enum chess { pawn, knight, bishop, rook, queen, king };

enum coins { penny, nickel, dime, quarter, half-dollar, dollar };
```

We'll see other examples in future programs.