Structures in C++

A structure is a collection of simple variables. The variables in a structure can be of different types: int, float, and so on. The data items in a structure are called the *members* of the structure. In fact, the syntax of a structure is almost identical to that of a class. A structure is a collection of data, while a class is a collection of both data and functions. Structures in C++ similar to *records* in Pascal.

A Simple Structure (user-defined data types)

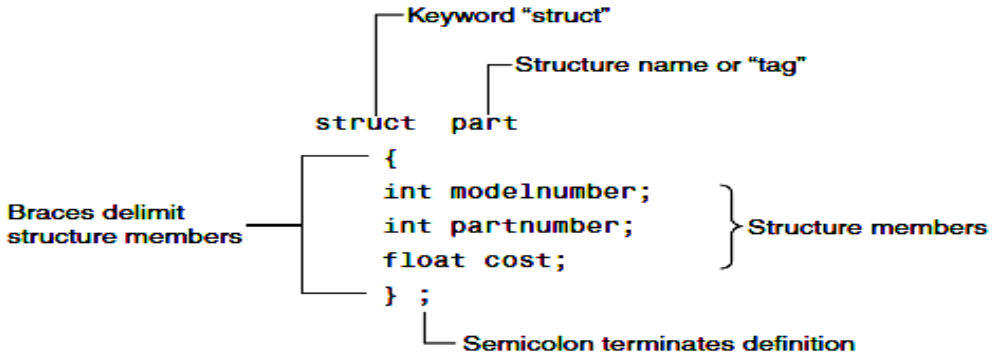
The company makes several kinds of widgets, so the widget model number is the first member of the structure.
The number of the part itself is the next member, and the final member is the part's cost. The next program defines the structure part, defines a structure variable of that type called part1. Assigns values to its members, and then displays these values.

```
#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
```

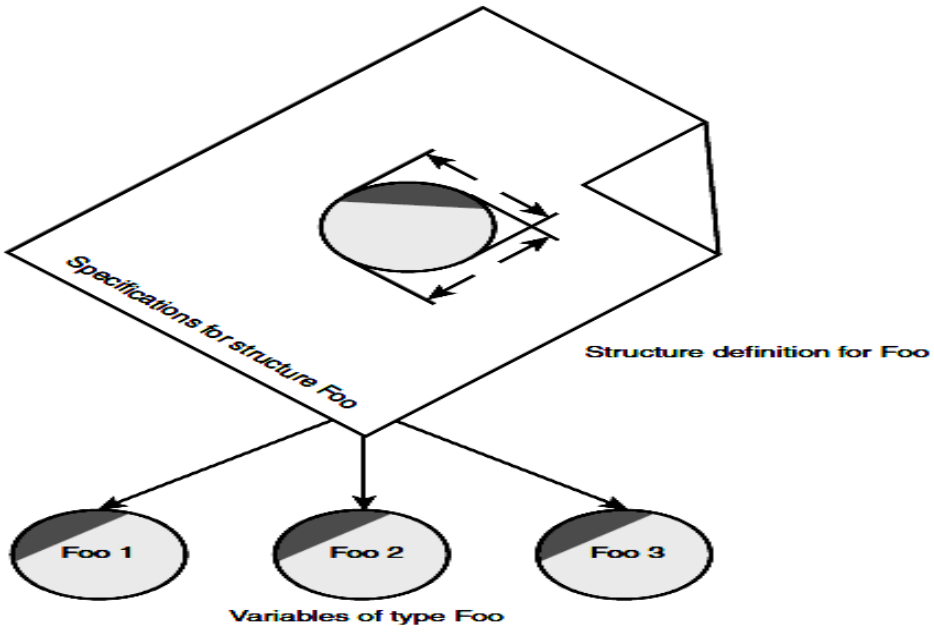
Syntax of the Structure Definition



Use of the Structure Definition

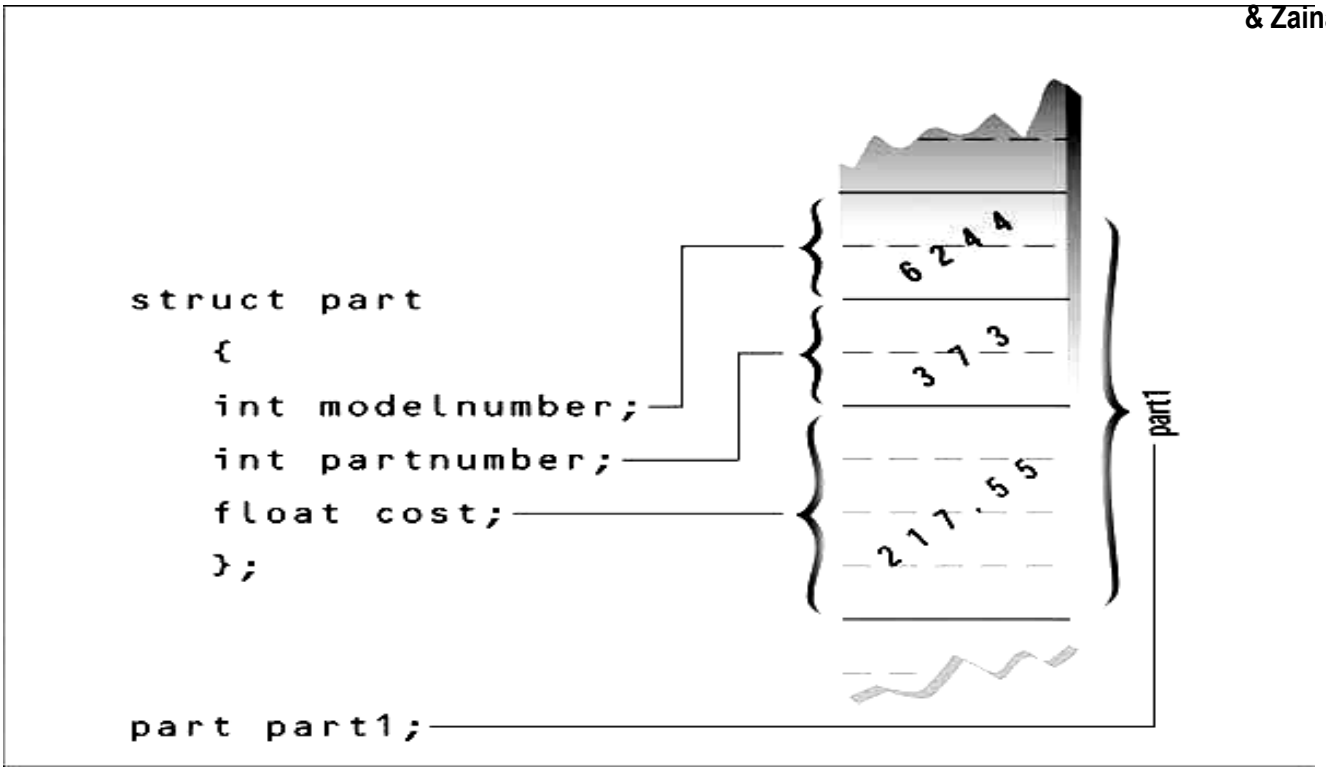
The structure definition serves only as a blueprint for the creation of variables of type part. It does not itself create any structure variables; that is, it does not set aside any space in memory or even name any variables.

This is unlike the definition of a simple variable, which does set aside memory. A structure definition is specification for how structure variables will look when they are defined. This is shown in Figure below:



Defining a Structure Variable

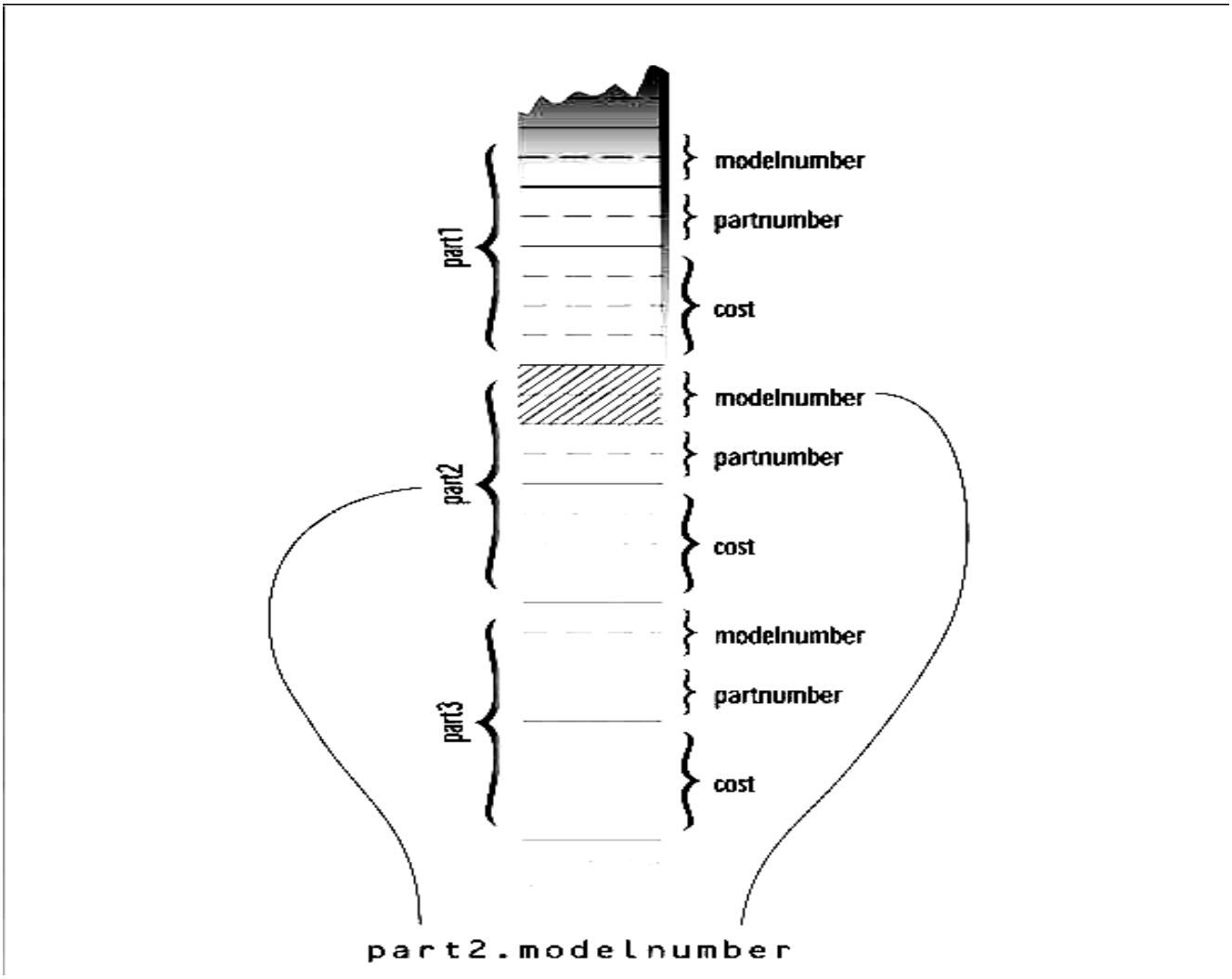
The first statement in main() part part1; defines a variable, called part1, of type structure part. This definition reserves space in memory for part1. Figure below shows how part1 looks inmemory.



Accessing Structure Members

members can be accessed using the dot operator.
part1.modelnumber = 6244;
The structure member is written in three parts:
1- the name of the structure variable (part1);
2- dot operator, which consists of a period (.);
3- member name (modelnumber).

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 below:



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.

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

```
#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;
}
```

Here's the output:

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

A Measurement Example

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, 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.

```
#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 }
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;}
```

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 previous program. 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.

Structures Within Structures

You can nest structures within other structures. In the next program we want to create a data structure that stores the dimensions of a typical room: its length and width.

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

```
#include <iostream>
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
```

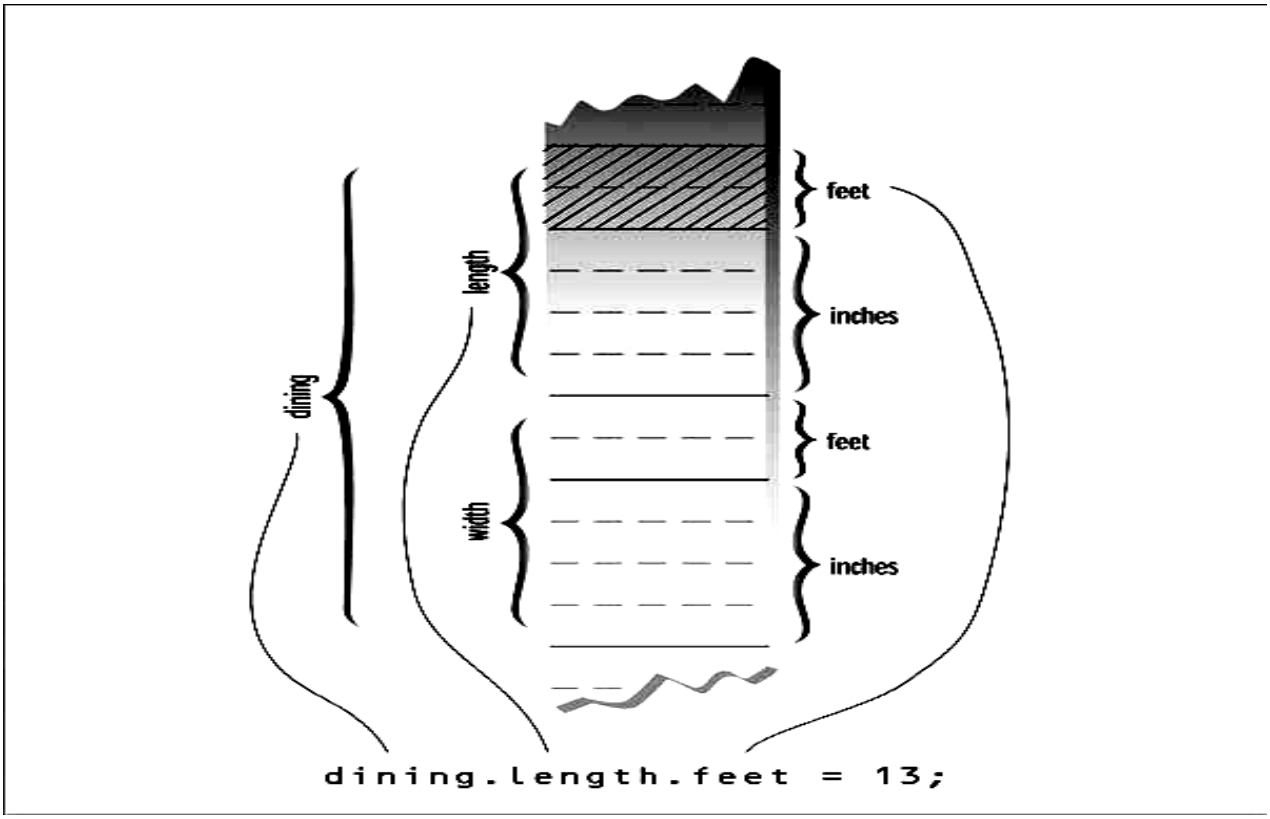
```
cout << "Dining room area is " << l * w << " square feet\n" ;
return 0;
}
```

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 below shows how this works.



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

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 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}

Exercises

1. A phone number, such as (212) 767-8900, can be thought of as having three parts: thearea code (212), the exchange (767), and the number (8900). Write a program that uses a structure to store these three parts of a phone number separately. Call the structure phone. Create two structure variables of type phone. Initialize one, and have the user input a number for the other one. Then display both numbers. The interchange might look like this:
Enter your area code, exchange, and number: 415 555 1212
My number is (212) 767-8900
Your number is (415) 555-1212
2. A point on the two-dimensional plane can be represented by two numbers: an x coordinate and a y coordinate. For example, (4,5) represents a point 4 units to the right of the vertical axis, and 5 units up from the horizontal axis. The sum of two points can be defined as a new point whose x coordinate is the sum of the x coordinates of the two points, and whose y coordinate is the sum of the y coordinates. Write a program that uses a structure called point to model a point. Define three points, and have the user input values to two of them. Then set the third point equal to the sum of the other two, and display the value of the new point. Interaction with the program might look like this:
Enter coordinates for p1: 3 4
Enter coordinates for p2: 5 7
Coordinates of p1+p2 are: 8, 11

3. Create a structure called Volume that uses three variables of type Distance (from the Program in the lecture above) to model the volume of a room. Initialize a variable of type Volume to specific dimensions, then calculate the volume it represents, and print out the result. To calculate the volume, convert each dimension from a Distance variable to a variable of type float representing feet and fractions of a foot, and then multiply the resulting three numbers.

Solutions to Exercises

1.

```
#include <iostream>
using namespace std;
struct phone
{
int area; //area code (3 digits)
int exchange; //exchange (3 digits)
int number; //number (4 digits)
};
int main()
{
phone ph1 = { 212, 767, 8900 }; //initialize phone number
phone ph2; //define phone number
// get phone no from user
cout << "\nEnter your area code, exchange, and number";
cout << "\n(Don't use leading zeros): ";
cin >> ph2.area >> ph2.exchange >> ph2.number;
cout << "\nMy number is " //display numbers << '(' << ph1.area << ") "
<< ph1.exchange << '-' << ph1.number;
cout << "\nYour number is "<< '(' << ph2.area << ") "<< ph2.exchange << '-' <<
ph2.number << endl;
return 0;
}
```

2.

```
#include <iostream>
using namespace std;
struct point
{
int xCo; //X coordinate
int yCo; //Y coordinate
};
int main()
{
point p1, p2, p3; //define 3 points
cout << "\nEnter coordinates for p1: "; //get 2 points
```

```
cin >> p1.xCo >> p1.yCo; //from user
cout << "Enter coordinates for p2: ";
cin >> p2.xCo >> p2.yCo;
p3.xCo = p1.xCo + p2.xCo; //find sum of
p3.yCo = p1.yCo + p2.yCo; //p1 and p2
cout << "Coordinates of p1+p2 are: " //display the sum
<< p3.xCo << ", " << p3.yCo << endl;
return 0;
}
```

3.

```
#include <iostream>
using namespace std;
struct Distance
{
int feet;
float inches;
};
struct Volume
{
Distance length;
Distance width;
Distance height;
};
int main()
{
float l, w, h;
Volume room1 = { { 16, 3.5 }, { 12, 6.25 }, { 8, 1.75 } };
l = room1.length.feet + room1.length.inches/12.0;
w = room1.width.feet + room1.width.inches /12.0;
h = room1.height.feet + room1.height.inches/12.0;
cout << "Volume = " << l*w*h << " cubic feet\n";
return 0; }
```

Enumerations

A different approach to defining your own data type is the enumeration they can simplify and clarify your programming.

Days of the Week Example

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:

```
#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
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;
}
```

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

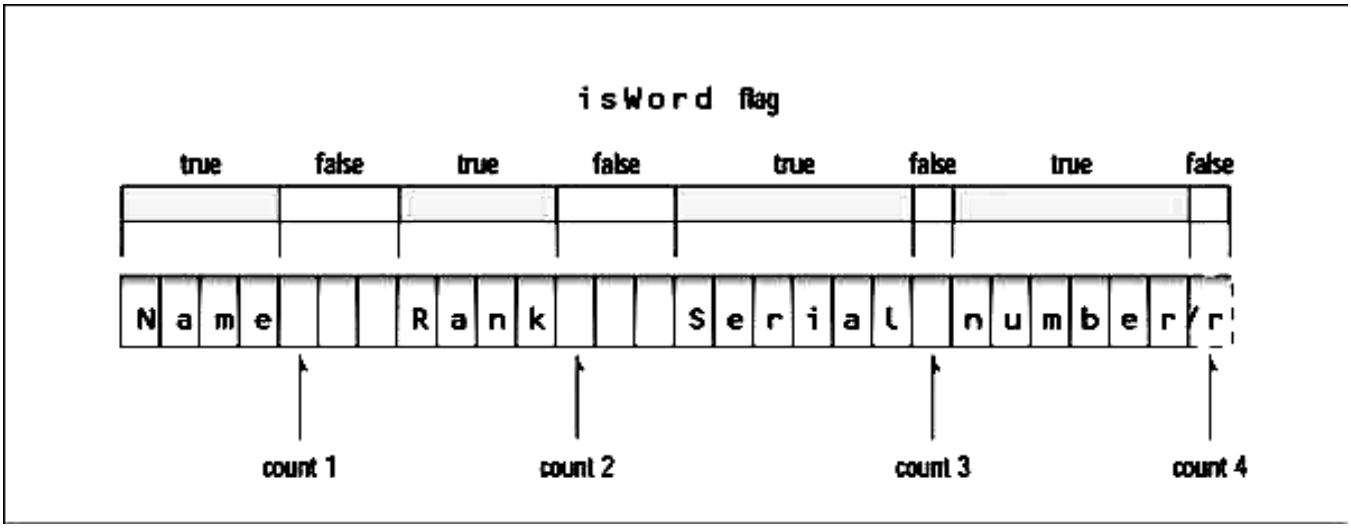
enum pets { cat, dog, hamster, canary, ocelot }; then it may not be clear what expressions like dog + canary or (cat < hamster) mean. Enumerations are treated internally as integers. This explains why you can perform arithmetic and relational operations on them.

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.

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 bellow:




```
#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;
}
```

EXERCISES

1. Create a structure called employee that contains two members: an employee number (type int) and the employee's compensation (in dollars; type float). Ask the user to fill in this data for three employees, store it in three variables of type struct employee, and then display the information for each employee.

2. Create a structure of type date that contains three members: the month, the day of the month, and the year, all of type int. (Or use day-month-year order if you prefer.)

Have the user enter a date in the format 12/31/2001, store it in a variable of type struct date, then retrieve the values from the variable and print them out in the same format.

3. We said earlier that C++ I/O statements don't automatically understand the data types of enumerations. Instead, the (>>) and (<<) operators think of such variables simply as integers.
You can overcome this limitation by using switch statements to translate between the user's way of expressing an enumerated variable and the actual values of the enumerated variable. For example, imagine an enumerated type with values that indicate an employee type within an organization:
enum etype { laborer, secretary, manager, accountant, executive,
researcher };
Write a program that first allows the user to specify a type by entering its first letter ('l', 's', 'm', and so on), then stores the type chosen as a value of a variable of type enum etype, and finally displays the complete word for this type.

Enter employee type (first letter only)
laborer, secretary, manager,
accountant, executive, researcher): a
Employee type is accountant.
You'll probably need two switch statements: one for input and one for output.

Functions

A function groups a number of program statements into a unit and gives it a name. This unit can then be invoked from other parts of the program.
The most important reason to use functions is to reduce program size. Any sequence of instructions that appears in a program more than once is being made into a function.

Eliminating the Declaration

The second approach to inserting a function into a program is to eliminate the function declaration and place the function definition (the function itself) in the listing before the first call to the function. For example, we could rewrite TABLE to produce TABLE2, in which the definition for starline() appears first.

Passing Arguments to Functions

An argument is a piece of data (an int value, for example) passed from a program to the function. Arguments allow a function to operate with different values, or even to do different things, depending on the requirements of the program calling it.

Passing Constants

As an example, let's suppose we decide that the starline() function in the last example is too rigid. Instead of a function that always prints 45 asterisks, we want a function that will print any character any number of times.
Here's a program, TABLEARG, that incorporates just such a function. We use arguments to pass the character to be printed and the number of times to print it.

```
#include <iostream>
using namespace std;
void repchar(char, int); //function declaration
int main()
{
repchar('-', 43); //call to function
cout << "Data type Range" << endl;
repchar('=', 23); //call to function
cout << "char -128 to 127" << endl
<< "short -32,768 to 32,767" << endl
<< "int System dependent" << endl
<< "double -2,147,483,648 to 2,147,483,647" << endl;
repchar('-', 43); //call to function
return 0;
}
//--------------------------------------------------------
// repchar()
// function definition
void repchar(char ch, int n)
{
for(int j=0; j<n; j++)
cout << ch;
cout << endl;
}
```

The calling program supplies arguments, such as '–' and 43, to the function.
The variables used within the function to hold the argument values are called parameters; in repchar() they are ch and n.

Passing Variables

In the TABLEARG example the arguments were constants: '–', 43, and so on.
Let's look at an example where variables, instead of constants, are passed as arguments.
This program, VARARG, incorporates the same repchar() function as did TABLEARG, but lets the user specify the character and the number of times it should be repeated.

```
#include <iostream>
using namespace std;
void repchar(char, int);
int main()
{
char chin;
int nin;
cout << "Enter a character: ";
cin >> chin;
```

```
cout << "Enter number of times to repeat it: ";
cin >> nin;
repchar(chin, nin);
return 0;
}
//--------------------------------------------------------------
void repchar(char ch, int n) //function declarator
{
for(int j=0; j<n; j++) //function body
cout << ch;
cout << endl;
}
```

Here's some sample interaction with VARARG:

```
Enter a character: +
Enter number of times to repeat it: 20
++++++++++++++++++++
```

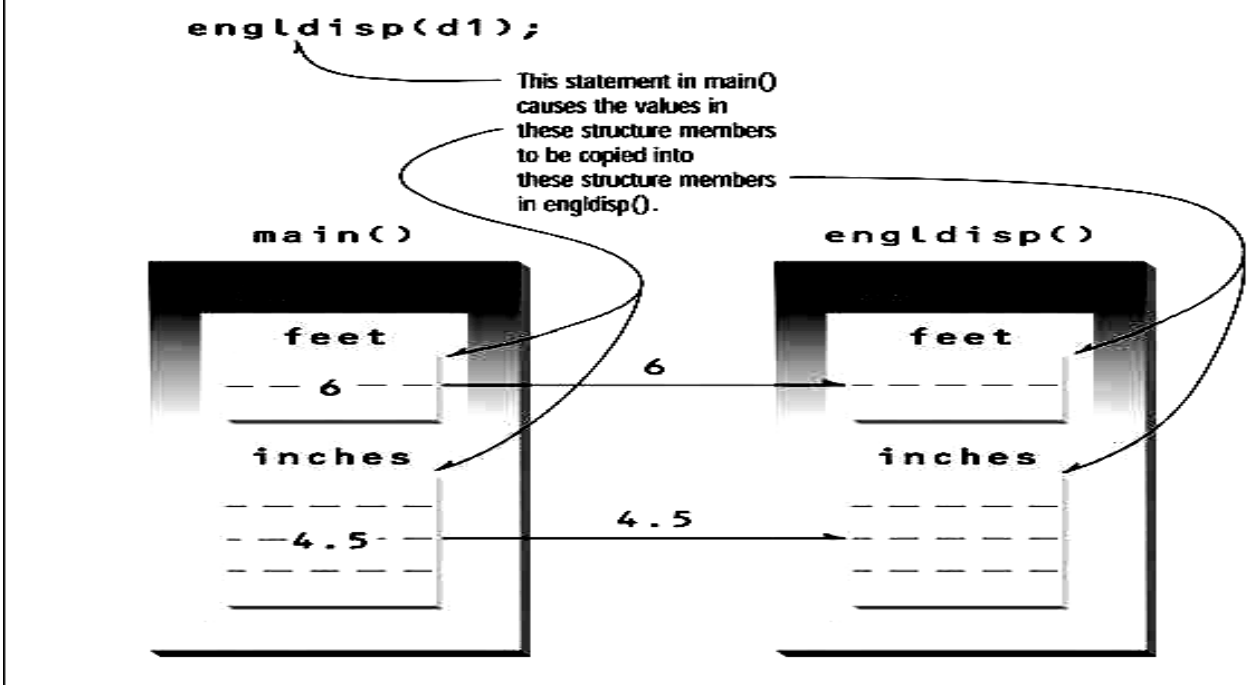
Structures as Arguments

Entire structures can be passed as arguments to functions.

Passing a Distance Structure example:

This example shows a function that uses an argument of type Distance. The main() part of this program accepts two distances in feet-and-inches format from the user, and places these values in two structures, d1 and d2. It then calls a function, disp(), that takes a Distance structure variable as an argument.

The purpose of the function is to display the distance passed to it in the standard format, such as 10'–2.25". Figure shows a structure being passed as an argument to a function.




```
#include <iostream>
using namespace std;
struct Distance //English distance
{ int feet;
float inches; };

void disp( Distance ); //declaration
int main()
{
Distance d1, d2; //define two lengths
cout << "Enter feet: "; cin >> d1.feet;
cout << "Enter inches: "; cin >> d1.inches;
cout << "\nEnter feet: "; cin >> d2.feet;
cout << "Enter inches: "; cin >> d2.inches;
cout << "\nd1 = "; disp(d1);
cout << "\nd2 = "; disp(d2);
cout << endl;
return 0;
}
void disp( Distance dd ) //parameter dd of type Distance
{
cout << dd.feet << "\'-" << dd.inches << "\"";
}
```

Here's some sample interaction with the program:

```
Enter feet: 6
Enter inches: 4
```

```
Enter feet: 5
Enter inches: 4.25

d1 = 6'-4"
d2 = 5'-4.25"
```

Returning Values from Functions

When a function completes its execution, it can return a single value to the calling program. Usually this return value consists of an answer to the problem the function has solved. The next example demonstrates a function that returns a weight in kilograms after being given a weight in pounds.

```
#include <iostream>
using namespace std;
float lbstokg(float); //declaration
int main()
{
float lbs, kgs;
cout << "\nEnter your weight in pounds: ";
cin >> lbs;
kgs = lbstokg(lbs);
cout << "Your weight in kilograms is " << kgs << endl;
return 0;
}
float lbstokg(float pounds)
{
float kilograms = 0.453592 * pounds;
return kilograms;
}
```

Here's some sample interaction with this program:

```
Enter your weight in pounds: 182
Your weight in kilograms is 82.553741
```

Returning Structure Variables

We've seen that structures can be used as arguments to functions. You can also use them as return values. Here's a program, RETSTRC that incorporates a function that adds variables of type Distance and returns a value of this same type:

```
#include <iostream>
using namespace std;
struct Distance //English distance
{
int feet;
float inches;};
Distance addengl(Distance, Distance); //declarations
void disp(Distance);
```

```
int main()
{ Distance d1, d2, d3; //define three lengths
cout << "\nEnter feet: "; cin >> d1.feet;
cout << "Enter inches: "; cin >> d1.inches;
cout << "\nEnter feet: "; cin >> d2.feet;
cout << "Enter inches: "; cin >> d2.inches;
d3 = addengl(d1, d2); //d3 is sum of d1 and d2
cout << endl;
disp(d1); cout << " + "; //display all lengths
disp(d2); cout << " = ";
disp(d3); cout << endl;
return 0; }

Distance addengl( Distance dd1, Distance dd2 )
{ Distance dd3; //define a new structure for sum
dd3.inches = dd1.inches + dd2.inches; //add the inches
dd3.feet = 0; //(for possible carry)
if(dd3.inches >= 12.0) //if inches >= 12.0,
{ //then decrease inches
dd3.inches -= 12.0; //by 12.0 and
dd3.feet++; }
dd3.feet += dd1.feet + dd2.feet; //add the feet
return dd3;  }
void disp( Distance dd )
{cout << dd.feet << "\'-" << dd.inches << "\"";}
```

The program asks the user for two lengths, in feet-and-inches format, adds them together by calling the function addengl(), and displays the results using the engldisp() function introduced in the LDISP program.
Here's some output from the program:

```
Enter feet: 4
Enter inches: 5.5
Enter feet: 5
Enter inches: 6.5
4'-5.5" + 5'-6.5" = 10'-0"
```

Reference Arguments

A reference provides a different name—for a variable. One of the most important uses for references is in passing arguments to functions. We've seen examples of function arguments passed by value. When arguments are passed by value, the called function creates a new variable of the same type as the argument and copies the argument's

value into it. As we noted, the function cannot access the original variable in the calling program, only the copy it created. Passing arguments by value is useful when the function does not need to modify the original variable in the calling program. In fact, it offers insurance that the function cannot harm the original variable. Passing arguments by reference uses a different mechanism. Instead of a value being passed to the function, a reference to the original variable, in the calling program, is passed. (It's actually the memory address of the variable that is passed.) An important advantage of passing by reference is that the function can access the actual variables in the calling program.

Among other benefits, this provides a mechanism for passing more than one value from the function back to the calling program. Suppose you have pairs of numbers in your program and you want to be sure that the smaller one always precedes the larger one. To do this you call a function, order(), which checks two numbers passed to it by reference and swaps the originals if the first is larger than the second.

```
#include <iostream>
using namespace std;
int main()
{void order(int&, int&); //prototype
int n1=99, n2=11; //this pair not ordered
int n3=22, n4=88; //this pair ordered
order(n1, n2); //order each pair of numbers
order(n3, n4);
cout << "n1=" << n1 << endl; //print out all numbers
cout << "n2=" << n2 << endl;
cout << "n3=" << n3 << endl;
cout << "n4=" << n4 << endl;
return 0;
}
void order(int& numb1, int& numb2) //orders two numbers
{
if(numb1 > numb2) //if 1st larger than 2nd,
{int temp = numb1; //swap them
numb1 = numb2;
numb2 = temp;}
}
```

In main() there are two pairs of numbers—the first pair is not ordered and the second pair is ordered. The order() function is called once for each pair, and then all the numbers are printed out. The output reveals that the first pair has been swapped while the second pair hasn't.

Here it is:

```
n1=11
n2=99
n3=22
```

n4=88

Passing Structures by Reference

You can pass structures by reference just as you can simple data types. A scale conversion involves multiplying a group of distances by a factor. If a distance is 6'–8", and a scale factor is 0.5, the new distance is 3'–4". Such a conversion might be applied to all the dimensions of a building to make the building shrink but remain in proportion.

```
#include <iostream>
using namespace std;
struct Distance
{ int feet;
float inches; };
void scale( Distance&, float );
void engldisp( Distance );
int main()
{ Distance d1 = { 12, 6.5 };
  Distance d2 = { 10, 5.5 };
  cout << "d1 = "; engldisp(d1);
  cout << "\nd2 = "; engldisp(d2);
  scale(d1, 0.5); //scale d1 and d2
  scale(d2, 0.25);
  cout << "\nd1 = "; engldisp(d1);
  cout << "\nd2 = "; engldisp(d2);
  cout << endl;
  return 0; }
void scale( Distance& dd, float factor)
 { float inches = (dd.feet*12 + dd.inches) * factor;
   dd.feet = (inches / 12);
   dd.inches = inches - dd.feet * 12;
}
void engldisp( Distance dd )
 {
 cout << dd.feet << "\'-" << dd.inches << "\"";
}
```

REFERST initializes two Distance variables—d1 and d2—to specific values, and displays them. Then it calls the scale() function to multiply d1 by 0.5 and d2 by 0.25.

Finally, it displays the resulting values of the distances. Here's the program's output:

```
d1 = 12'-6.5"
d2 = 10'-5.5"
d1 = 6'-3.25"
d2 = 2'-7.375"
```

Here are the two calls to the function scale():

```
scale(d1, 0.5);
```

```
scale(d2, 0.25);
```

Exercises

*1. Refer to the CIRCAREA program in Chapter 2, "C++ Programming Basics." Write a function called circarea() that finds the area of a circle in a similar way. It should take an argument of type float and return an argument of the same type. Write a main() function that gets a radius value from the user, calls circarea(), and displays the result.

*2. Raising a number n to a power p is the same as multiplying n by itself p times. Write a function called power() that takes a double value for n and an int value for p, and returns the result as a double value. Use a default argument of 2 for p, so that if this argument is omitted, the number n will be squared. Write a main() function that gets values from the user to test this function.

*3. Write a function called zeroSmaller() that is passed two int arguments by reference and then sets the smaller of the two numbers to 0. Write a main() program to exercise this function.

*4. Write a function that takes two Distance values as arguments and returns the larger one. Include a main() program that accepts two Distance values from the user, compares them, and displays the larger. (See the RETSTRC program for hints.)

Exercise

5- rite a function called swap() that interchanges two int values passed to it by the calling program. (Note that this function swaps the values of the variables in the calling program, not those in the function.) You'll need to decide how to pass the arguments. Create a main() program to exercise the function.

Solutions to Exercises

1.

```
// ex5_1.cpp
// function finds area of circle
#include <iostream>
using namespace std;
float circarea(float radius);
int main()
{
double rad;
cout << "\nEnter radius of circle: ";
cin >> rad;
cout << "Area is " << circarea(rad) << endl;
return 0;
}
//--------------------------------------------------------------
float circarea(float r)
{
const float PI = 3.14159;
return r * r * PI;
}
```

2.

```
#include <iostream>
```

```
using namespace std;
double power( double n, int p=2); //p has default value 2
int main()
{double number, answer;
int pow;
char yesorno;
cout << "\nEnter number: "; //get number
cin >> number;
cout << "Want to enter a power (y/n)? ";
cin >> yesorno;
if( yesorno == 'y' ) //user wants a non-2 power?
{cout << "Enter power: ";
cin >> pow;
answer = power(number, pow); //raise number to pow}
else
answer = power(number); //square the number
cout << "Answer is " << answer << endl;
return 0;}
double power( double n, int p )
{double result = 1.0; //start with 1
for(int j=0; j<p; j++) //multiply by n
result *= n; //p times
return result;}
```

3.

```
#include <iostream>
using namespace std;
int main()
{
void zeroSmaller(int&, int&);
int a=4, b=7, c=11, d=9;
zeroSmaller(a, b);
zeroSmaller(c, d);
cout << "\na=" << a << " b=" << b
<< " c=" << c << " d=" << d;
return 0;
}
// sets the smaller of two numbers to 0
void zeroSmaller(int &first, int &second)
{
if( first < second )
first = 0;
else
second = 0;
}
```

4.

```
#include <iostream>
```

```
using namespace std;
struct Distance // English distance
{int feet;
float inches;};

Distance bigengl(Distance, Distance); //declarations
void engldisp(Distance);
int main()
{Distance d1, d2, d3; //define three lengths
cout << "\nEnter feet: "; cin >> d1.feet;
cout << "Enter inches: "; cin >> d1.inches;
//get length d2 from user
cout << "\nEnter feet: "; cin >> d2.feet;
cout << "Enter inches: "; cin >> d2.inches;
d3 = bigengl(d1, d2); //d3 is larger of d1 and d2
//display all lengths
cout << "\nd1="; engldisp(d1);
cout << "\nd2="; engldisp(d2);
cout << "\nlargest is "; engldisp(d3); cout << endl;
return 0;}
//-----------------------------------------------------------
Distance bigengl( Distance dd1, Distance dd2 )
{
if(dd1.feet > dd2.feet) //if feet are different, return
return dd1; //the one with the largest feet
if(dd1.feet < dd2.feet)
return dd2;
if(dd1.inches > dd2.inches) //if inches are different,
return dd1; //return one with largest
else //inches, or dd2 if equal
return dd2;
}
//-----------------------------------------------------------
// engldisp()
// display structure of type Distance in feet and inches
void engldisp( Distance dd )
{
cout << dd.feet << "\'-" << dd.inches << "\"";
}
```