1.8 Enumerations

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

1.8.1 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.h>
//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:

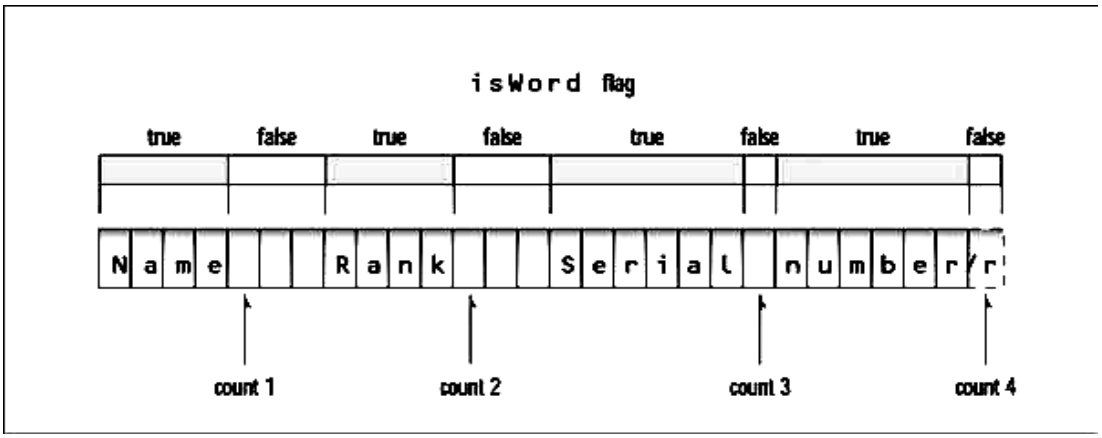
```
Days between = 3
day1 comes before day2
```

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.

1.8.2 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.h>
#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. 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.

1.9 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.

1.9.1 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.

1.9.2 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.

1.9.2.1 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.h>
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.

1.9.2.2 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.h>
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:

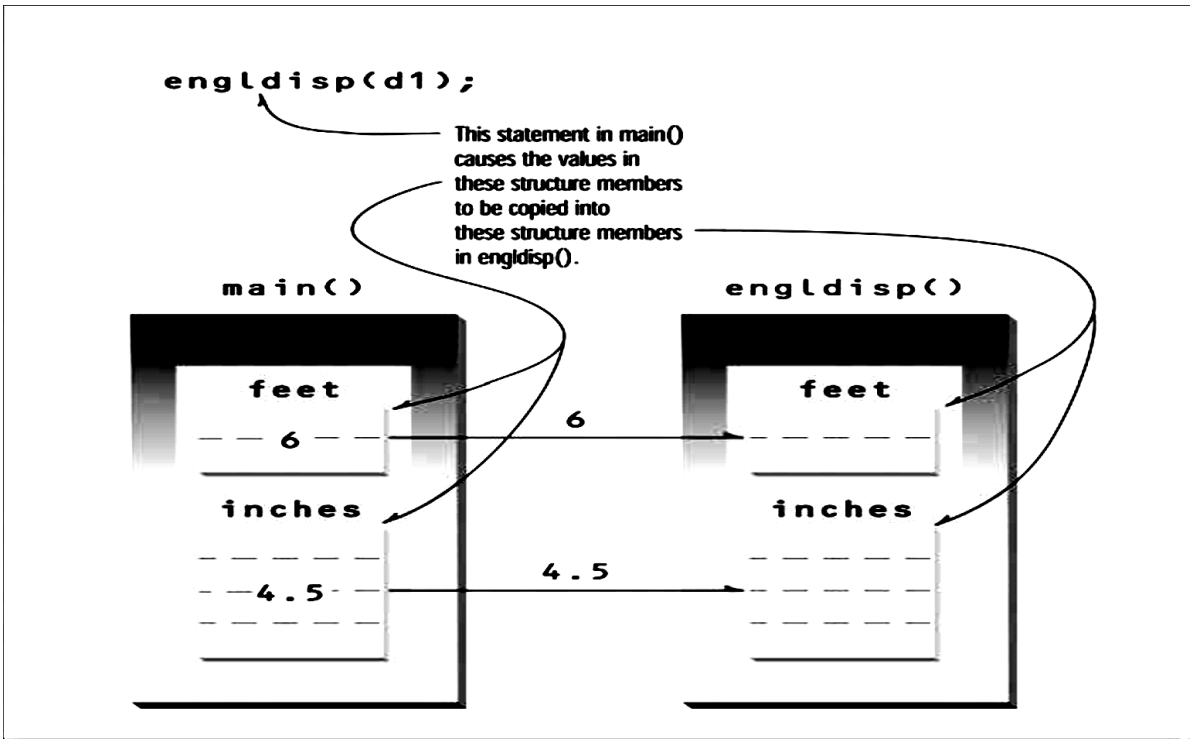
```
Enter a character: +
Enter number of times to repeat it: 20
++++++++++++++++++++
```

1.9.3 Structures as Arguments

Entire structures can be passed as arguments to functions.

1.9.3.1 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.h>
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"
```

1.9.4 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.h>
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

1.9.5 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.h>
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"
```

1.9.6 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.h>
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
```

1.9.10 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.h>
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.

Solutions to Exercises

1.

```
// ex5_1.cpp
// function finds area of circle
#include <iostream.H>
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.h>

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;
}
```