3.6 User-Defined Data Types

Structures and Classes

We have used user-defined data types such as **struct** and **union** in C. While these data type are legal in C++, some more features have been added to make them suitable for objec oriented programming. C++ also permits us to define another user-defined data type know as **class** which can be used, just like any other basic data type, to declare variables. Th class variables are known as objects, which are the central focus of object-oriente programming. More about these data types is discussed later in Chapter 5.

Enumerated Data Type

An enumerated data type is another user-defined type which provides a way for attachin names to numbers, thereby increasing comprehensibility of the code. The **enum** keywor (from C) automatically enumerates a list of words by assigning them values 0,1,2, and so on This facility provides an alternative means for creating symbolic constants. The syntax of a **enum** statement is similar to that of the **struct** statement. Examples:

```
enum shape{circle, square, triangle};
enum colour{red, blue, green, yellow};
enum position{off, on};
```

The enumerated data types differ slightly in C++ when compared with those in ANSI C. I C++, the tag names **shape, colour**, and **position** become new type names. By using thes tag names, we can declare new variables. Examples:

```
shape ellipse;          // ellipse is of type shape
colour background;      // background is of type colour
```

ANSI C defines the types of **enums** to be **ints**. In C++, each enumerated data type retains its own separate type. This means that C++ does not permit an **int** value to be automatically converted to an **enum** value. Examples:

```
colour background = blue;          // allowed
colour background = 7;             // Error in C++
colour background = (colour) 7;    // OK
```

However, an enumerated value can be used in place of an **int** value.

```
int c = red;      // valid, colour type promoted to int
```

By default, the enumerators are assigned integer values starting with 0 for the first enumerator, 1 for the second, and so on. We can over-ride the default by explicitly assigning integer values to the enumerators. For example,

```
enum colour{red, blue=4, green=8};
enum colour{red=5, blue, green};
```

are valid definitions. In the first case, **red** is 0 by default. In the second case, **blue** is 6 and **green** is 7. Note that the subsequent initialized enumerators are larger by one than their predecessors.

C++ also permits the creation of anonymous **enums** (i.e., **enums** without tag names). Example:

```
enum{off, on};
```

Here, **off** is 0 and **on** is 1. These constants may be referenced in the same manner as regular constants. Examples:

```
int switch_1 = off;
int switch_2 = on;
```

In practice, enumeration is used to define symbolic constants for a **switch** statement. Example:

```
enum shape
{
   circle,
   rectangle,
   triangle
};

int main()
{
cout << "Enter shape code:";
int code;
cin >> code;
while(code >= circle && code <= triangle)
{
        switch(code)
```

```
{
        case circle:
        ......
        ......
        break;
        case rectangle:
        ......
        ......
                break;
                case triangle:
                ......
                ......
                break;
        }
        cout << "Enter shape code:";
        cin >> code;
    }
    cout << "BYE \n";

    return 0;
}
```