12.2 C-Style Enumerations

The declaration

```c
enum Color {RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET};
```

creates a new type named `Color` whose values are the seven colors listed between the curly braces. Because the valid values are explicitly listed or enumerated in the declaration, this kind of type is called an enumeration.

**ENUMERATION DECLARATIONS**

The declaration of an enumeration must:

1. Provide a name for the enumeration, which becomes the name of a new type
2. Explicitly list all of the values (called enumerators) of this new type

In the example above, `Color` is the name of the enumeration, and its enumerators are the identifiers

```
RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET
```

When the compiler encounters such a declaration, it performs an object-to-integer mapping, associating the integer 0 with the first identifier in this list, the integer 1 with the second, and so on. Thus, for the preceding declaration, the compiler makes the following associations:

```

    RED   ORANGE   YELLOW   GREEN   BLUE    INDIGO   VIOLET
  0     1     2     3     4     5      6
```

As another example, the declaration

```c
enum Gender {FEMALE, MALE};
```

declares a new type `Gender` whose values are the identifiers `FEMALE` and `MALE`; the compiler will associate the integer 0 with `FEMALE` and the integer 1 with `MALE`. Similarly, the declaration

```c
enum HandTool {HAMMER, PLIERS, SAW, SCREWDRIVER};
```

constructs a new type `HandTool` whose values are `HAMMER`, `PLIERS`, `SAW`, and `SCREWDRIVER`, and associates the integers 0, 1, 2, and 3 with these identifiers, respectively. By contrast, neither of the declarations

```c
enum Zipcodes {12531, 14405, 21724, 30081};  // ERROR!
    C-, D+, D, D-, "FAIL"};
```

is a valid enumeration, because each contains items that are not valid identifiers.

C++ also allows the programmer to specify explicitly the values given to the enumerators. For example, the declaration
enum NumberBase { BINARY = 2, 
    OCTAL = 8, 
    DECIMAL = 10, 
    HEX = 16, HEXADECIMAL = 16; 
}

associates the identifiers BINARY, OCTAL, DECIMAL, HEX, and HEXADECIMAL with the values 2, 8, 10, 16, and 16, respectively. Because each enumerator is a power of 2, each has a 1 at a different position in its binary representation. Such enumerators are called bit masks, and make it possible to efficiently store a boolean value such as an iostream status attribute using only a single bit of memory.

Similarly, if we wished to have the values 1, 2, ..., 7 associated with the seven colors given earlier (instead of 0 through 6), we could use the declaration

```cpp
enum Color { RED = 1, ORANGE = 2, YELLOW = 3, GREEN = 4, 
    BLUE = 5, INDIGO = 6, VIOLET = 7; 
```
or more compactly,

```cpp
enum Color { RED = 1, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET; 
```
because the integer associated with an enumerator is, by default, one more than the integer associated with the preceding enumerator. The iostream library uses an enumeration declaration something like

```cpp
enum Flag { GOOD_BIT = 1, BAD_BIT, FAIL_BIT = 4, EOF_BIT = 8; 
```
which associates 1 with GOOD_BIT, 2 with BAD_BIT, 4 with FAIL_BIT, and 8 with EOF_BIT.¹

These examples illustrate the flexibility of C++—the integers associated with the names need not be distinct nor must they be given in ascending order, although it is good programming style to do so.

The general form of an enumeration declaration is as follows:

```
Enumeration Declaration Statement
Form:
    enum TypeName { List }; 

where:
    TypeName is an identifier naming a new type; and

List is a list of the values for the new type, separated by commas, each of which is a valid

IDENTIFIER

or an initialization expression of the form

IDENTIFIER = integer_constant
```
Purpose:

Define a new data type whose values are the identifiers in List. Each identifier is associated with an integer as follows:

If an item in List has the form IDENTIFIER = integer_constant, then integer_constant is associated with IDENTIFIER;

otherwise if it is the first item in the list,

0 is associated with the IDENTIFIER;

otherwise,

1 + (the integer associated with the preceding identifier) is associated with the IDENTIFIER.

Because the compiler essentially treats an enumeration as a series of constant integer declarations, we use the same uppercase naming convention for enumerators that we use for constant objects.

DEFINING ENUMERATION OBJECTS

To illustrate how enumerations are used, consider the following expansion of enumeration Color:

```cpp
enum Color {
    COLOR_UNDERFLOW = -1, // too-low indicator
    RED, ORANGE, YELLOW, GREEN, // 0-3
    BLUE, INDIGO, VIOLET, // 4-6
    COLOR_OVERFLOW, // too-high indicator
    NUMBER_OF_COLORS = 7};
```

Here, we added the identifiers COLOR_UNDERFLOW and COLOR_OVERFLOW as values to indicate out-of-range errors. These values can be used to keep from “falling off the ends of the list.” We also added the identifier NUMBER_OF_COLORS, whose value is the number of values in the list, because this count is often useful.

If it is worthwhile to define a new type, it is usually worth taking the time to store that type in a library so that it can be easily reused. We thus store this declaration of type Color in a header file Color.h so that programs can include it and avoid reinventing the wheel.

Given this type, we can declare a Color object named theColor:

```cpp
Color theColor;
```

Enumeration objects can also be initialized when they are declared:

```cpp
Color theColor = YELLOW;
```

USING ENUMERATIONS

In addition to defining enumeration objects, an enumeration can be used as the index of an array. For example, suppose we define colorArray as follows:

```cpp
double colorArray[NUMBER_OF_COLORS] = {0.0};
```

This definition builds the object colorArray as a fixed-size array with index values 0 through 6. Because the C++ compiler treats the identifiers RED through VIOLET as the integer values 0 through 6, we can visualize colorArray as follows:
colorArray

<table>
<thead>
<tr>
<th>0.0</th>
<th>0.0</th>
<th>0.0</th>
<th>0.0</th>
<th>0.0</th>
<th>0.0</th>
<th>0.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RED</td>
<td>ORANGE</td>
<td>YELLOW</td>
<td>GREEN</td>
<td>BLUE</td>
<td>INDIGO</td>
<td>VIOLET</td>
</tr>
</tbody>
</table>

The Color enumerators can then be used with the subscript operator to access the array elements.

In the same way, an extra enumerator like NUMBER_OF_COLORS can be used to provide a vector<T> with an initial size:

```cpp
vector<double> colorVector(NUMBER_OF_COLORS);
``` 

This defines colorVector as a varying-sized object, initially with NUMBER_OF_COLORS elements:

colorVector

<table>
<thead>
<tr>
<th>0.0</th>
<th>0.0</th>
<th>0.0</th>
<th>0.0</th>
<th>0.0</th>
<th>0.0</th>
<th>0.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RED</td>
<td>ORANGE</td>
<td>YELLOW</td>
<td>GREEN</td>
<td>BLUE</td>
<td>INDIGO</td>
<td>VIOLET</td>
</tr>
</tbody>
</table>