

Definition of Enumerator Constants

Enumerators are considered defined immediately after their initializers; therefore, they can be used to initialize succeeding enumerators. The following example defines an enumerated type that ensures that any two enumerators can be combined with the OR operator:

```
enum FileOpenFlags
{
    OpenReadOnly = 1,
    OpenReadWrite = OpenReadOnly << 1,
    OpenBinary    = OpenReadWrite << 1,
    OpenText      = OpenBinary    << 1,
    OpenShareable = OpenText      << 1
};
```

In this example, the preceding enumerator initializes each succeeding enumerator.
Built on Thursday, May 11, 2000

enum

enum [tag] {enum-list} [declarator]; // for definition of enumerated type

enum tag declarator; // for declaration of variable of type tag

The enum keyword specifies an enumerated type.

An enumerated type is a user-defined type consisting of a set of named constants called enumerators. By default, the first enumerator has a value of 0, and each successive enumerator is one larger than the value of the previous one, unless you explicitly specify a value for a particular enumerator. Enumerators needn't have unique values. The name of each enumerator is treated as a constant and must be unique within the scope where the enum is defined. An enumerator can be promoted to an integer value. However, converting an integer to an enumerator requires an explicit cast, and the results are not defined.

In C, you can use the enum keyword and the tag to declare variables of the enumerated type. In C++, you can use the tag alone.

In C++, enumerators defined within a class are accessible only to member functions of that class unless qualified with the class name (for example, class_name::enumerator). You can use the same syntax for explicit access to the type name (class_name::tag).

For related information, see class and struct.

Example

```
// Example of the enum keyword
enum Days          // Declare enum type Days
{
    saturday,       // saturday = 0 by default
    sunday = 0,     // sunday = 0 as well
    monday,         // monday = 1
    tuesday,        // tuesday = 2
    wednesday,      // etc.
    thursday,
    friday
} today;           // Variable today has type Days

int tuesday;       // Error, redefinition of tuesday

enum Days yesterday; // Legal in C and C++
Days tomorrow;      // Legal in C++ only
```

```
yesterday = monday;
```

```
int i = tuesday;    // Legal; i = 2
yesterday = 0;      // Error; no conversion
yesterday = (Days)0; // Legal, but results undefined
```

C++ Enumeration Declarations

```
enum [tag] {enum-list} [declarator];
// for definition of enumerated type
```

```
enum tag declarator; // for declaration of variable of type tag
```

An enumeration is a user-defined type consisting of a set of named constants called enumerators. By default, the first enumerator has a value of 0, and each successive enumerator is one larger than the value of the previous one, unless you explicitly specify a value for a particular enumerator. Enumerators need not have unique values within an enumeration. The name of each enumerator is treated as a constant and must be unique within the scope where the enum is defined. An enumerator can be promoted to an integer value. However, converting an integer to an enumerator requires an explicit cast, and the results are not defined.

Enumerated types are valuable when an object can assume a known and reasonably limited set of values. Consider the example of the suits from a deck of cards:

```
class Card
{
public:
    enum Suit
    {
        Diamonds,
        Hearts,
        Clubs,
        Spades
    };
    // Declare two constructors: a default constructor,
    // and a constructor that sets the cardinal and
    // suit value of the new card.
    Card();
    Card( int CardInit, Suit SuitInit );

    // Get and Set functions.
    int  GetCardinal();    // Get cardinal value of card.
    int  SetCardinal();    // Set cardinal value of card.
    Suit GetSuit();        // Get suit of card.
    void SetSuit(Suit new_suit); // Set suit of card.
    char *NameOf();        // Get string representation of card.
private:
    Suit suit;
    int  cardinalValue;
};

// Define a postfix increment operator for Suit.
inline Card::Suit operator++( Card::Suit &rs, int )
{
    return rs = (Card::Suit)(rs + 1);
}
```

The preceding example defines a class, *Card*, that contains a nested enumerated type, *Suit*. To create a pack of cards in a program, use code such as:

```
Card *Deck[52];
```

```
int j = 0;
```

```
for( Card::Suit curSuit = Card::Diamonds; curSuit <= Card::Spades;
    curSuit++ )
    for( int i = 1; i <= 13; ++i )
        Deck[j++] = new Card( i, curSuit );
```

In the preceding example, the type `Suit` is nested; therefore, the class name (`Card`) must be used explicitly in public references. In member functions, however, the class name can be omitted.

In the first segment of code, the postfix increment operator for `Card::Suit` is defined. Without a user-defined increment operator, `curSuit` could not be incremented. For more information about user-defined operators, see *Overloaded Operators* in Chapter 12.

Consider the code for the `NameOf` member function (a better implementation is presented later):

```
char* Card::NameOf() // Get the name of a card.
{
    static char szName[20];
    static char *Numbers[] =
    { "1", "2", "3", "4", "5", "6", "7", "8", "9",
      "10", "Jack", "Queen", "King"
    };
    static char *Suits[] =
    { "Diamonds", "Hearts", "Clubs", "Spades" };

    if( GetCardinal() < 13 )
        strcpy( szName, Numbers[GetCardinal()] );

    strcat( szName, " of " );

    switch( GetSuit() )
    {
        // Diamonds, Hearts, Clubs, and Spades do not need explicit
        // class qualifier.
        case Diamonds: strcat( szName, "Diamonds" ); break;
        case Hearts:   strcat( szName, "Hearts" );   break;
        case Clubs:    strcat( szName, "Clubs" );    break;
        case Spades:   strcat( szName, "Spades" );   break;
    }

    return szName;
}
```

An enumerated type is an integral type. The identifiers introduced with the `enum` declaration can be used wherever constants appear. Normally, the first identifier's value is 0 (`Diamonds`, in the preceding example), and the values increase by one for each succeeding identifier. Therefore, the value of `Spades` is 3.

Any enumerator in the list, including the first one, can be initialized to a value other than its default value. Suppose the declaration of `Suit` had been the following:

```
enum Suit
{
    Diamonds = 5,
    Hearts,
    Clubs = 4,
    Spades
};
```

Then the values of Diamonds, Hearts, Clubs, and Spades would have been 5, 6, 4, and 5, respectively. Note that 5 is used more than once.

The default values for these enumerators simplify implementation of the NameOf function:

```
char* Card::NameOf() // Get the name of a card.
{
    static char szName[20];
    static char *Numbers[] =
    { "1", "2", "3", "4", "5", "6", "7", "8", "9",
      "10", "Jack", "Queen", "King"
    };
    static char *Suits[] =
    { "Diamonds", "Hearts", "Clubs", "Spades" };

    if( GetCardinal() < 13)
        strcpy( szName, Numbers[GetCardinal()] );

    strcat( szName, " of " );

    strcat( szName, Suits[GetSuit()] );

    return szName;
}
```

The accessor function GetSuit returns type Suit, an enumerated type. Because enumerated types are integral types, they can be used as arguments to the array subscript operator. (For more information, see Subscript Operator in Chapter 4.)

Example

```
#include <iostream.h>

enum Days          // Declare enum type Days
{
    saturday,       // saturday = 0 by default
    sunday = 0,     // sunday = 0 as well
    monday,         // monday = 1
    tuesday,        // tuesday = 2
    wednesday,      // etc.
    thursday,
    friday
};

void main() {
    enum Days today = sunday;
    switch (today) {
        case 1:
            cout << "\nIt's Monday" << endl;
            break;
        default:
            cout << "\nNot Monday" << endl;
    }
}
```

In C, the enum keyword is required to declare a variable of type enumeration. In C++, the enum keyword can be omitted. For example, given the Days enumeration from the code above:

```
Days tomorrow;    // Legal in C++ only
```

Grammar

```

enum-name:
    identifier
enum-specifier:
    enum identifieropt { enumerator-listopt }
enumerator-list:
    enumerator-definition
    enumerator-list , enumerator-definition
enumerator-definition:
    enumerator
    enumerator = constant-expression
enumerator:
    identifier

```

Conversions and Enumerated Types

Because enumerated types are integral types, any enumerator can be converted to another integral type by integral promotion. Consider this example:

```

enum Days
{
    Sunday,
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
};

int i;
Days d = Thursday;

i = d; // Converted by integral promotion.
cout << "i = " << i << "\n";

```

However, there is no implicit conversion from any integral type to an enumerated type. Therefore (continuing with the preceding example), the following statement is in error:

```

d = 6; // Erroneous attempt to set d to Saturday.

```

Assignments such as this, where no implicit conversion exists, must use a cast to perform the conversion:

```

d = (Days)6; // Explicit cast-style conversion to type Days.
d = Days( 4 ); // Explicit function-style conversion to type Days.

```

The preceding example shows conversions of values that coincide with the enumerators. There is no mechanism that protects you from converting a value that does not coincide with one of the enumerators. For example:

```

d = Days( 967 );

```

Some such conversions may work. However, there is no guarantee that the resultant value will be one of the enumerators. Additionally, if the size of the enumerator is too small to hold the value being converted, the value stored may not be what you expect.