

## 4.8 Enumeraciones

### §1 Sinopsis

Las **variables enumeradas**, **enumeraciones** o más abreviadamente **enum** (palabra reservada), son un tipo especial de variables que tienen la propiedad de que su rango de valores es un conjunto de constantes enteras [1] denominadas **constantes de enumeración**, a cada una de las cuales se le asigna un valor (📖 3.2.3g). Para mayor legibilidad del código estos valores del rango se identifican por nemónicos. Por ejemplo, la declaración:

```
enum dia { DOM, LUN, MART, MIER, JUEV, VIER, SAB } diaX;
```

establece un tipo **enum** al que se identifica por **dia**. Las variables de este tipo pueden adoptar un conjunto de seis valores enteros 0, 1, 2, 3, 4, 5, 6 (**enumeradores**) representados por los nemónicos [2] **DOM, LUN, ..., SAB**. Además se define una variable **diaX** de este tipo (variable enumerada).

En el ejemplo anterior, la variable **diaX** puede adoptar siete valores enteros (del 0 al 6). El programa los identifica con los nemónicos ya señalados.

💡 Cada enumeración distinta constituye un tipo de enumerando diferente. Ejemplo:

```
enum Calificacion {APTO, NO-APTO} c1;  
enum Evolucion {SUBE, BAJA, IGUAL} c2;
```

**c1** y **c2** son objetos de tipo distinto.

**§2** Puede omitirse la palabra clave **enum** si **dia** no es identificador de nada más en el mismo ámbito. El compilador sabe que se trata de una variable tipo **enum** por la sintaxis de la propia declaración. Por ejemplo, es correcto:

```
Dia { DOM, LUN, MART, MIER, JUEV, VIER, SAB } diaX;
```

El identificador **Dia** es la etiqueta opcional del tipo, y puede ser usada en subsecuentes declaraciones de variables del mismo tipo:

```
enum Dia laborable, festivo; // declara dos nuevas  
variables
```

aunque también se podría omitir el especificador **enum** (el compilador ya sabe que **Dia** es de tipo **enum**):

```
Dia laborable, festivo; // equivalente al anterior
```

**§3** Como ocurre con las declaraciones de estructuras y uniones, si no existen otras variables del mismo tipo, puede omitirse la etiqueta con lo que tenemos un **tipo anónimo**:

```
enum { DOM, LUN, MART, MIER, JUEV, VIER, SAB } diaX;
```

También aquí (como ocurre con las estructuras y uniones), el inconveniente de declarar un tipo anónimo es que después no podemos volver a declarar otra variable del mismo tipo. 💡 Observe que las expresiones que siguen no son equivalentes!!

```
enum diaX { DOM, LUN, MART, MIER, JUEV, VIER, SAB };  
enum { DOM, LUN, MART, MIER, JUEV, VIER, SAB } diaX;
```

**§4** En C, una variable enumerada puede recibir cualquier valor de tipo **int** sin que se realice ninguna otra comprobación; en cambio C++ es más fuertemente tipado en este sentido (📖 2.2), y a una variable enumerada solo se le puede asignar uno de sus enumeradores. Ejemplo:

```
diaX = LUN;           // Ok.
diaX = 1;             // Ilegal, a pesar que LUN == 1
```

**Nota:** El conmutador **-b** del compilador Borland C++ (📖 1.4.3) controla la opción "Considerar las enumeraciones como enteros". Se refiere a que esta opción controla el almacenamiento que asignará el compilador a las variables enumeradas. Cuando se activa la opción, el compilador reserva una palabra (4 bytes en un compilador de 32 bits) para las **variables enumeradas**. El valor por defecto es ON, que significa que se les asignará el espacio de un entero, lo que por otra parte es lo máximo que pueden necesitar porque las **constantes de enumeración** son están forzosamente en el rango de los enteros.

Si se desactiva (opción **-b-**), el compilador comprueba el rango de los enumeradores y reserva el espacio estrictamente necesario según el criterio de la tabla adjunta (ver al respecto: Representación interna y rango 📖 2.2.4); fuera de estos rangos asigna 32 bits:

<u>Rango de valores</u>	<u>espacio</u>	<u>tipo equivalente</u>
0 a 255	8 bits	<b>unsigned char</b>
-128 a 127	8 bits	<b>signed char</b>
0 a 65,535	16 bits	<b>unsigned short</b>
-32,768 a 32,767	16 bits	<b>signed short</b>

Por su parte, los identificadores utilizados en la lista de enumeradores son implícitamente del tipo **signed char**, **unsigned char** o **int**, dependiendo de los valores asignados. Si todos los valores pueden ser representados con un **signed** o **unsigned char**, ese es el tipo de todos ellos.

Hay que recordar que, al ser mostradas por el dispositivo de salida, estas variables son promovidas a enteros y este es el valor mostrado. En el ejemplo anterior **MART** es mostrado como 2. En C++ se puede conseguir escribir el tipo **enum** sobrecargando adecuadamente el operador **<<**, (ver sobrecarga de operadores). La opción por defecto es que el tipo **enum** es promovido a entero y se imprime el valor resultante.

**§5** Como puede verse, las enumeraciones proporcionan un modo muy flexible de asociar nombres con valores, algo que también puede hacerse con los **#define** (📖 4.9.10b), sin embargo las enumeraciones (aún de un solo elemento) presentan ventajas [3] entre las que podemos señalar:

- Los valores pueden ser auto-generados
- Las variables de enumeradas ofrecen la posibilidad de comprobación.
- El depurador puede ser capaz de imprimir los valores de los enumeradores en su forma simbólica (lo que facilita grandemente las comprobaciones).

**§6** Los tipos **enum** pueden aparecer en cualquier sitio donde sea permitido un entero. Ejemplo:

```
enum dias { LUN, MAR, MIER, JUEV, VIER, SAB, DOM } diaX;
enum dias diapago;
typedef enum dias DIAS;
DIAS *diaptr;
int i = MIER;
diaX = LUN;           // Ok.
*diaptr = diaX;       // Ok.
LUN = MIER;           // ILEGAL: LUN es una constante!
```

## §7 Visibilidad

Las etiquetas de los **enum** comparten el mismo espacio de nombres que los de **estructuras** y **uniones**. Los identificadores de los enumeradores comparten el mismo espacio que los identificadores de las variables ordinarias; de aquí se deduce que los nombres de los enumeradores de las distintas enumeraciones, tienen que ser diferentes (los valores no tienen que serlo ni aún en la misma enumeración).

**Ejemplo:**

```
int dom = 11;
{
    enum dias { dom, lun, mar, mier, juev, vier, sab } diaX;
    /* el enumerador dom oculta otra declaracion de int dom */
    struct dias { int i, j;}; // ILEGAL: identificador dias ya
usado
    double mar;               // ILEGAL: redefinicion de mar
}
dom = 12;                     // int dom vuelve a ser visible
```

Los enumeradores declarados dentro de una clase tienen la visibilidad de la clase.

**Ejemplo:**

```
class Date {
public:
    std::string day;
    std::string month;
    std::string year;
    enum fomrat { US, SP, UK, FR, DE };
    int fm;
};
...
void foo() {
    Date d1;
    d1.fm = Date::US;
    ...
}
```

**Nota:** Es pertinente hacer aquí una observación: junto con las constantes estáticas enteras, los enumeradores son las únicas propiedades que pueden inicializarse dentro de la definición de la clase ([📖 4.11.2a](#))

## §8 Asignaciones a tipo enum

En el compilador Borland C++ 5.5, las reglas para expresiones que contengan tipos **enum** pueden forzarse a ser más estrictas activando el conmutador **-A** (que significa

ANSI C++ estricto), con lo que el compilador refuerza las reglas con mensajes de error. Así, al asignar un entero a una variable **enum** (como en el ejemplo) produciría un error.

```
enum color { Rojo, Verde, Azul };
int f() {
    color c;
    c = 0;      // Incorrecto: a 'c' solo se le puede asignar
    Rojo, Verde o Azul
    return c;
}
```

En estos casos de compilación estricta, se obtiene también un error cuando se pasa un entero como parámetro a una función que espera un **enum** o viceversa. Estudie el ejemplo propuesto teniendo en cuenta que el resultado de la expresión `flag1|flag2` (OR inclusivo entre bits [4.9.3](#)) es un entero.

```
#include <iostream.h>
enum En { flag1 = 0x01, flag2 = 0x02 };
int f1(En) {                               // L.3:
    return (flag1 + 1);
}
void f2() {
    int x = f1(flag1|flag2);               // L.7:
    cout << "El valor es: " << x << endl;
}

int main () {                             // =====
    f2();
    return 0;
}
```

Aquí `En` es un tipo **enum**; los enumeradores `flag1` y `flag2` son constantes de enumeración de valores 1 y 2 respectivamente, que han definidas en formato hexadecimal ([3.2.3b](#)).

Al intentar compilar se produce un error: `Cannot convert 'int' to 'En' in function f2()` ya que la invocación a `f1` en L7 espera un **enum** tipo `En` (como indica el prototipo de L3), pero se le pasa un **int**, ya que como se ha apuntado, la expresión `flag1|flag2` es un entero.

Para arreglarlo, aplicamos un modelado de tipo ([4.9.9](#)) al argumento pasado a la función:

```
En (flag1|flag2)
```

que lo transforma en el tipo exigido, con lo que la compilación se realiza sin novedad.

```
#include <iostream.h>
enum En { flag1 = 0x01, flag2 = 0x02 };
int f1(En) {                               // L.3:
    return (flag1 + 1);
}
void f2() {
    int x = f1(En (flag1|flag2) );         // L.7:
    cout << "El valor es: " << x << endl;
}

int main () {
    f2();
}
```

Salida:

```
El valor es: 2
```


✓ **Tema relacionado:** Sobrecarga de enumeraciones ( [4.9.18h](#)).

▲ [Inicio](#).

---

[1] Aunque comparten características con los enteros, constituyen un tipo especial de variable con identidad propia y necesitan un **cast** para que les pueda ser asignado un entero; ellas mismas son automáticamente promovidas a enteros cuando es necesario (ver más adelante las diversas opciones que ofrece el compilador al respecto).

[2] Es una costumbre universalmente aceptada que los identificadores que representan constantes se escriban con mayúsculas, incluyendo por supuesto los nombres de los enumeradores (que representan constantes).

[3] De hecho el propio Stroustrup señala en el prólogo de su  [TC++PL](#) : “Macros (#define) are almost never necessary in C++; use **const** or **enum** to define manifest constants”.

---

[ [Atrás](#) ] [ [Siguiente](#) ] [ [Índice](#) ]

Copyright © 2000-2005 Zator Systems. **NO** software patents