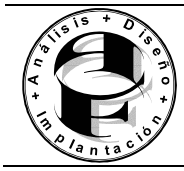



# Fundamentos de la Programación

## TEMA 6: SENTENCIAS DE CONTROL

### OBJETIVOS

En principio, las sentencias de un programa en C se ejecutan *secuencialmente*, esto es, cada una a continuación de la anterior empezando por la primera y acabando por la última. Esta tipo de programación es adecuado para resolver problemas sencillos. Sin embargo, para la resolución de problemas de tipo general se necesita la capacidad de controlar cuáles son las sentencias que se han de ejecutar en cada momento.

El lenguaje C/C++ dispone de varias estructuras de control para modificar este flujo secuencial de la ejecución, que se agrupan en tres grandes categorías: **secuencia**, **selección** y **repetición**.

En este tema comenzaremos viendo algunos operadores avanzados que utilizaremos en las sentencias selectivas y repetitivas que se estudiarán inmediatamente después.

El objetivo principal de este tema es que el alumno aprenda a utilizar las estructuras básicas de la programación estructurada, y que las aplique en la resolución de problemas sencillos.

### CONTENIDOS

*Introducción*
*1. Operadores avanzados*
*2. Expresiones*
*3. Reglas de precedencia y asociatividad*
*4. Sentencias*
*5. Estructuras selectivas*
*6. Estructuras repetitivas*
*7. Ejercicios propuestos*

### Bibliografía

- Joyanes Aguilar, J. “Programación en C++. Algoritmos, estructuras de datos y Objetos”. Capítulo 3,4 y 5. Ed. McGraw-Hill.
- Pont, M.J. “Software Engineering with C++ and CASE Tools”. Capítulo 3. Ed. Addison-Wesley.

# 1. Operadores avanzados

## 1.1. Operadores incrementales

Los ***operadores incrementales*** (++) y (--) son operadores unarios que incrementan o disminuyen ***en una unidad*** el valor de la variable a la que afectan. Estos operadores pueden ir inmediatamente delante o detrás de la variable. Si preceden a la variable, ésta es incrementada antes de que el valor de dicha variable sea utilizado en la expresión en la que aparece. Si es la variable la que precede al operador, la variable es incrementada después de ser utilizada en la expresión. A continuación se presenta un ejemplo de estos operadores:

```
i = 2;
j = 2;
m = i++;    // despues de ejecutarse esta sentencia m=2 e i=3
n = ++j;    // despues de ejecutarse esta sentencia n=3 y j=3
```

Estos operadores son muy utilizados. Es importante entender muy bien por qué los resultados **m** y **n** del ejemplo anterior son diferentes. Veamos algunos ejemplos más.

**Ejemplo 6.1. demostración del funcionamiento de los operadores de incremento/decremento**

```
#include <stdio.h>
// Prueba de operadores ++ y --
main()
{
   int m = 45, n = 75;
   printf("m = %d, n = %d\n", m, n);
   ++m;
   --n;
   printf("m = %d, n = %d\n", m, n);
   m++;
   n--;
   printf("m = %d, n = %d\n", m, n);
}
```

**Salida:**

```
m = 45 n = 75
m = 46 n = 74
m = 47 n = 73
```

**Ejemplo 3.2. Diferencias entre operadores de preincremento y postincremento.**

```
#include <stdio.h>
// prueba de operadores ++, --
main()
{
   int m = 99, n;
   n = ++m;
   printf("m = %d, n = %d\n", m, n);
   n = m++;
   printf("m = %d, n = %d\n", m, n);
   printf("m = %d\n", m++);
   printf("m = %d\n", ++m);
   return 0;
}
```

**Salida:**

```
m = 100, n = 100
m = 101, n = 100
m = 101
m = 103
```

**Ejemplo 3.3.Orden de evaluación no predecible en expresiones**

```
#include <stdio.h>
void main()
{
   int n = 5, t;
   t = ++n *, --n;
   printf("n= %d, t = %d\n", n, t);
   printf("%d %d %d\n" , ++n, ++n, ++n);
}
```

**Salida:**

```
n= 5, t = 25
```

```
8 7 6
```

Aunque parece que aparentemente el resultado de `t` será 30, en realidad es 25, debido a que en la asignación de `t`, `n` se incrementa a 6 y a continuación se decrementa a 5 antes de que se evalúe el operador producto, calculando `5 * 5`. Por último, las tres subexpresiones se evalúan de derecha a izquierda, el resultado será `8 7 6`, al contrario de `6 7 8`, que es lo que aparentemente se producirá.

## 1.2. Operadores relacionales

Este es un apartado especialmente importante para todas aquellas personas sin experiencia en programación. Una característica imprescindible de cualquier lenguaje de programación es la de ***considerar alternativas***, esto es, la de proceder de un modo u otro según se cumplan o no ciertas condiciones. Los ***operadores relacionales*** permiten estudiar si se cumplen o no esas condiciones. Así pues, estos operadores producen un resultado u otro según se cumplan o no algunas condiciones que se verán a continuación.

En el lenguaje natural, existen varias palabras o formas de indicar si se cumple o no una determinada condición. En inglés estas formas son (***yes***, ***no***), (***on***, ***off***), (***true***, ***false***), etc. En Informática se ha hecho bastante general el utilizar la última de las formas citadas: (***true***, ***false***). Si una condición se cumple, el resultado es ***true***; en caso contrario, el resultado es ***false***.

En C un 0 representa la condición de ***false***, y cualquier número distinto de 0 equivale a la condición ***true***. Cuando el resultado de una expresión es ***true*** y hay que asignar un valor concreto distinto de cero, por defecto se toma un valor unidad.

ANSI C++ soporta el tipo ***bool***, que admite dos literales: ***false*** y ***true***.

Los ***operadores relacionales*** de C son los siguientes:

| Operador | Significado | Ejemplo |
|---|---|---|
| == | Igual que | a == b |
| < | Menor que | a < b |
| > | Mayor que | a > b |
| <= | Menor o igual que | a <= b |
| >= | Mayor o igual que | a >= b |
| != | Distinto que: | a != b |

Todos los ***operadores relacionales*** son operadores ***binarios*** (tienen dos operandos), y su forma general es la siguiente:

```
expresion1 op expresion2
```

donde ***op*** es uno de los operadores (==, <, >, <=, >=, !=). El funcionamiento de estos operadores es el siguiente: se evalúan **expresion1** y **expresion2**, y se comparan los valores resultantes. *Si la condición representada por el operador relacional se cumple, el resultado es 1; si la condición no se cumple, el resultado es 0*.

A continuación se incluyen algunos ejemplos de estos operadores aplicados a constantes:

```
(2==1) // false, porque la condición no se cumple
(3<=3) // true, porque la condición se cumple
(3<3)  // false, porque la condición no se cumple
(1!=1) // false, porque la condición no se cumple
```

## 1.3. Operadores lógicos

Los ***operadores lógicos*** son operadores binarios que permiten combinar los resultados de los operadores relacionales, comprobando que se cumplen simultáneamente varias condiciones, que se cumple una u otra, etc. Los operadores lógicos del lenguaje C++ aparecen en la tabla 3.1.

Tabla 3.1. Operadores lógicos

| Operador | Operación |
|---|---|
| && | AND. Da como resultado el valor lógico 1 si ambos operandos son distintos de cero. Si uno de ellos es cero el resultado es el valor lógico 0. Si el primer operando es igual a cero, el segundo operando no es evaluado. |
| \|\| | OR. El resultado es 0 si ambos operandos son 0. Si uno de los operandos tiene un valor distinto de cero, el resultado es 1. Si el primer operando es distinto de cero, el segundo operando no es evaluado |
| ! | NOT. El resultado es 0 si el operando tiene un valor distinto de cero, y 1 en caso contrario |

Su forma general es la siguiente:

```
expresion1 || expresion2
expresion1 && expresion2
```

Los operadores **&&** y **||** se pueden combinar entre sí —quizás agrupados entre paréntesis—, dando a veces un código de más difícil interpretación. Por ejemplo:

```
(2==1) || (-1==-1)                        // el resultado es 1
(2==2) && (3==-1)                         // el resultado es 0
((2==2) && (3==3)) || (4==0)              // el resultado es 1
((6==6) || (8==0)) && ((5==5) && (3==2)) // el resultado es 0
```

## 1.4. Operadores de manipulación de bits

Una de las razones por las que C y C++ se han hecho tan populares es que ofrecen muchos operadores de manipulación de bits a bajo nivel.

Los operadores de manipulación o tratamiento de bits (*bitwise*) ejecutan operaciones lógicas sobre cada uno de los bits de los operandos. Estos operandos solamente pueden ser variables y constantes de tipo *char*, int y *long*.

| Operador | Operación |
|---|---|
| & | AND lógico bit a bit |
| \| | OR lógico bit a bit |
| ^ | XOR (OR exclusivo) lógico bit a bit |
| ~ | Complemento a uno (inversión de todos los bits) |
| << | Desplazamiento de bits a la izquierda |
| >> | Desplazamiento de bits a la derecha |

Ejemplo: vamos a aplicar diferentes operadores a los números 10 y 12

| AND, & | OR, \| | XOR, ^ |
|---|---|---|
| 10 → 1 0 1 0<br>& & & & | 10 → 1 0 1 0<br>\| \| \| \| | 10 → 1 0 1 0<br>^ ^ ^ ^ |

```
12 → 1 1 0 0          12 → 1 1 0 0          12 → 1 1 0 0
     -------               -------               -------
     1 0 0 0  → 8          1 1 1 0  → 14         1 0 0 1  → 9
```

## 1.5. Otros operadores

Además de los operadores vistos hasta ahora, el lenguaje C++ dispone de otros operadores.

### 1.5.1. El operador condicional

El operador condicional es un operador ternario que que se utiliza en expresiones condicionales, que tienen la forma:

```
operando1 ? operando2 : operando3
```

La expresión operando1 debe ser de tipo entero, real o puntero. La evaluación se realiza de la siguiente forma:

Si el resultado de la evaluación de operando1 es distinta de cero (true), el resultado de la operación condicional es operando2.

Si el resultado de la evaluación de operando1 es cero (false), el resultado de la operación condicional es operando3.

Ejemplo:

```
mayor = (a > b) ? a : b;    // mayor de a y b
```

En este ejemplo, si `a>b` entonces `mayor = a`; en caso contrario, `mayor = b`.

### 1.5.2. El operador `sizeof()`

Este es el operador de C++ con el nombre más largo. Puede parecer una función, pero en realidad es un operador. La finalidad del operador **sizeof()** es devolver el tamaño, en ***bytes***, del tipo de variable introducida entre los paréntesis. Recuérdese que este tamaño depende del compilador y del tipo de computador que se está utilizando, por lo que es necesario disponer de este operador para producir código portable. Por ejemplo:

```
var_1 = sizeof(double); // var_1 contiene el tamaño de una variable dou-
ble
```

El operador *sizeof* se denomina también *operador en tiempo de compilación*, ya que el compilador sustituye cada ocurrencia de *sizeof* en el programa por un valor entero sin signo.

**Ejemplo 3.4. Obtención del tamaño de una variable en coma flotante**

```
#include <stdio.h>
main()
{
   printf << "El tamaño de variables de coma flotante ";
   printf << "de este ordenador es: " << sizeof(float) << '\n';
   return 0;
}
```

**Salida:**

```
El tama±o de variables de coma flotante de este ordenador es: 4
```

### 1.5.3. El operador coma

El operador *coma* permite combinar dos o más expresiones separadas por comas en una sola línea. Se evalúa primero la expresión de la izquierda, y luego las restantes expresiones de izquierda a derecha. La expresión más a la derecha determina el resultado global. Los operandos de este operador son expresiones, y tiene la forma general:

```
expresion = expresion_1, expresion_2, ..., expresion_n;
```

En este caso, **expresion_1** se evalúa primero, luego se evalúa **expresion_2**, y así sucesivamente. El resultado global es el valor de la última expresión, es decir de **expresion_n**. Este es el operador de menor prioridad de todos los operadores de C++. Como se explicará más adelante, su uso más frecuente es para introducir expresiones múltiples en la sentencia ***for***.

## 2. Expresiones

Ya han aparecido algunos ejemplos de expresiones del lenguaje C en las secciones precedentes. Una **expresión** es una combinación de variables y/o constantes, y operadores. La expresión es equivalente al resultado que proporciona al aplicar sus operadores a sus operandos. Por ejemplo, 1+5 es una expresión formada por dos *operandos* (1 y 5) y un *operador* (el +); esta expresión es equivalente al valor 6, lo cual quiere decir que allí donde esta expresión aparece en el programa, en el momento de la ejecución es evaluada y sustituida por su resultado. Una expresión puede estar formada por otras expresiones más sencillas, y puede contener paréntesis de varios niveles agrupando distintos términos. En C++ existen distintos tipos de expresiones.

### 2.1. Expresiones aritméticas

Están formadas por variables y/o constantes, y distintos operadores aritméticos e incrementales (+, -, *, /, %, ++, --). Como se ha dicho, también se pueden emplear paréntesis de tantos niveles como se desee, y su interpretación sigue las normas aritméticas convencionales. Por ejemplo, la solución de la ecuación de segundo grado:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

se escribe, en C en la forma:

```
x=(-b+sqrt((b*b)-(4*a*c)))/(2*a);
```

donde, estrictamente hablando, sólo lo que está a la derecha del operador de asignación (=) es una expresión aritmética. El conjunto de la variable que está a la izquierda del signo (=), el operador de asignación, la expresión aritmética y el carácter (;) constituyen una ***sentencia***. En la expresión anterior aparece la llamada a la ***función de librería sqrt()***, que tiene como ***valor de retorno*** la raíz cuadrada de su único ***argumento***. En las expresiones se pueden introducir espacios en blanco entre operandos y operadores; por ejemplo, la expresión anterior se puede escribir también de la forma:

```
x = (-b + sqrt((b * b) - (4 * a * c)))/(2 * a);
```

### 2.2. Expresiones lógicas

Los elementos con los que se forman estas expresiones son ***valores lógicos***; *verdaderos* (***true***, o distintos de 0) y *falsos* (***false***, o iguales a 0), y los ***operadores lógicos*** **||**, **&&** y **!**. También se pueden emplear los ***operadores relacionales*** (<, >, <=, >=, ==, !=) para producir estos valores lógicos a partir de valores numéricos. Estas expresiones equivalen siempre a un valor 1 (***true***) o a un valor 0 (***false***). Por ejemplo:

```
a = ((b>c)&&(c>d))||((c==e)||(e==b));
```

donde de nuevo la ***expresión lógica*** es lo que está entre el operador de asignación (=) y el (;). La variable **a** valdrá 1 si **b** es mayor que **c** *y* **c** mayor que **d**, ó si **c** es igual a **e** ó **e** es igual a **b**.

## 2.3. Expresiones generales

Una de las características más importantes (y en ocasiones más difíciles de manejar) del C es su flexibilidad para combinar expresiones y operadores de distintos tipos en una expresión que se podría llamar *general*, aunque es una expresión absolutamente ordinaria de C.

Recuérdese que el resultado de una expresión lógica es siempre un valor numérico (un 1 ó un 0); esto permite que cualquier expresión lógica pueda aparecer como sub-expresión en una expresión aritmética. Recíprocamente, cualquier valor numérico puede ser considerado como un valor lógico: ***true*** si es distinto de 0 y ***false*** si es igual a 0. Esto permite introducir cualquier expresión aritmética como sub-expresión de una expresión lógica. Por ejemplo:

```
(a - b*2.0) && (c != d)
```

A su vez, *el operador de asignación* (=), además de introducir un nuevo valor en la variable que figura a su izda, *deja también este valor disponible para ser utilizado* en una expresión más general. Por ejemplo, supóngase el siguiente código que inicializa a 1 las tres variables **a**, **b** y **c**:

```
a = b = c = 1;
```

que equivale a:

```
a = (b = (c = 1));
```

En realidad, lo que se ha hecho ha sido lo siguiente. En primer lugar se ha asignado un valor unidad a **c**; el resultado de esta asignación es también un valor unidad, que está disponible para ser asignado a **b**; a su vez el resultado de esta segunda asignación vuelve a quedar disponible y se puede asignar a la variable **a**.

# 3. Reglas de precedencia y asociatividad

El resultado de una expresión depende del orden en que se ejecutan las operaciones. El siguiente ejemplo ilustra claramente la importancia del orden. Considérese la expresión:

```
3 + 4 * 2
```

Si se realiza primero la suma (3+4) y después el producto (7*2), el resultado es 14; si se realiza primero el producto (4*2) y luego la suma (3+8), el resultado es 11. Con objeto de que el resultado de cada expresión quede claro e inequívoco, es necesario definir las reglas que definen el orden con el que se ejecutan las expresiones de C. Existe dos tipos de reglas para determinar este orden de evaluación: las reglas de ***precedencia*** y de ***asociatividad***. Además, el orden de evaluación puede modificarse por medio de paréntesis, pues *siempre se realizan primero las operaciones encerradas en los paréntesis más interiores*. Los distintos operadores de C se ordenan según su distinta ***precedencia*** o prioridad; para operadores de la misma precedencia o prioridad, en algunos el orden de ejecución es de izquierda a derecha, y otros de derecha a izquierda (se dice que *se asocian* de izda a dcha, o de dcha a izda). A este orden se le llama ***asociatividad***.

En la tabla 6.2 se muestra la precedencia —en orden decreciente— y la asociatividad de los operadores de C++. En dicha Tabla se incluyen también algunos operadores que no han sido vistos hasta ahora.

Tabla 6.2. Precedencia y asociatividad de los operadores de C.

| Precedencia | Asociatividad |
|---|---|
| `() [] -> .` | izda a dcha |
| `++ -- ! sizeof (tipo) +(unario) -(unario) *(indir.) &(dirección)` | dcha a izda |
| `* / %` | izda a dcha |

| | |
|---|---|
| `+ -` | izda a dcha |
| `< <= > >=` | izda a dcha |
| `== !=` | izda a dcha |
| `&&` | izda a dcha |
| `\|\|` | izda a dcha |
| `?:` | dcha a izda |
| `= += -= *= /=` | dcha a izda |
| `, (operador coma)` | izda a dcha |

En la tabla anterior se indica que el operador (*) tiene precedencia sobre el operador (+). Esto quiere decir que, en ausencia de paréntesis, el resultado de la expresión 3+4*2 es 11 y no 14. Los operadores binarios (+) y (-) tienen igual precedencia, y asociatividad de izda a dcha. Eso quiere decir que en la expresión,

```
a-b+d*5.0+u/2.0;       // (((a-b)+(d*5.0))+(u/2.0))
```

el orden de evaluación es el indicado por los paréntesis en la parte derecha de la línea (Las últimas operaciones en ejecutarse son las de los paréntesis más exteriores).

# 4. Sentencias

Las ***expresiones*** de C son unidades o componentes elementales de unas entidades de rango superior que son las ***sentencias***. Las sentencias son unidades completas, ejecutables en sí mismas. Ya se verá que muchos tipos de sentencias incorporan expresiones aritméticas, lógicas o generales como componentes de dichas sentencias.

## 4.1. Sentencias simples

Una sentencia simple es una expresión de algún tipo terminada con un carácter (;). Un caso típico son las declaraciones o las sentencias aritméticas. Por ejemplo:

```
float real;
espacio = espacio_inicial + velocidad * tiempo;
```

## 4.2. Sentencia vacía ó nula

En algunas ocasiones es necesario introducir en el programa una sentencia *que ocupe un lugar, pero que no realice ninguna tarea*. A esta sentencia se le denomina ***sentencia vacía*** y consta de un simple carácter (;). Por ejemplo:

```
;
```

## 4.3. Sentencias compuestas o bloques

Muchas veces es necesario poner varias sentencias en un lugar del programa donde debería haber una sola. Esto se realiza por medio de ***sentencias compuestas***. Una sentencia compuesta es un conjunto de declaraciones y de sentencias agrupadas dentro de llaves **{...}**. También se conocen con el nombre de ***bloques***. Una sentencia compuesta puede incluir otras sentencias, simples y compuestas. Un ejemplo de sentencia compuesta es el siguiente:

```
{
   int i = 1, j = 3, k;
   double masa;
   masa = 3.0;
   k = y + j;
}
```

Las sentencias compuestas se utilizarán con mucha frecuencia en el tema siguiente, al introducir las sentencias que permiten modificar el flujo de control del programa.

# 5. Estructuras selectivas

En principio, las sentencias de un programa en C/C++ se ejecutan *secuencialmente*, esto es, cada una a continuación de la anterior empezando por la primera y acabando por la última. Esta tipo de programación es adecuado para resolver problemas sencillos. Sin embargo, para la resolución de problemas de tipo general se necesita la capacidad de controlar cuáles son las sentencias que se han de ejecutar en cada momento.

El lenguaje C/C++ dispone de varias estructuras de control para modificar este flujo secuencial de la ejecución, que se agrupan en tres grandes categorías: **secuencia**, **selección** y **repetición**.

En este tema se consideran las estructuras selectivas o condicionales —sentencias `if` y `switch`— que controlan si una sentencia o lista de sentencias se ejecutan en función del cumplimiento o no de una condición.

## 5.1. La sentencia `if`

Esta sentencia de control permite ejecutar o no una sentencia simple o compuesta según se cumpla o no una determinada condición. Esta sentencia tiene la siguiente forma general:

```
if (expresion)
   Acción;
```

La sentencia `if` funciona de la siguiente manera. Cuando se alcanza la sentencia `if` dentro de un programa, se evalúa la expresión entre parentesis que viene a cntinuación de `if`. Si **expresion** es verdadera, se ejecuta **Acción**; en caso contrario no se ejecuta **Acción**. En cualquier caso, la ejecución del programa continúa con la siguiente sentencia del programa. La Figura 6.1 muestra el diagrama de flujo correspondiente a una sentencia `if` simple.

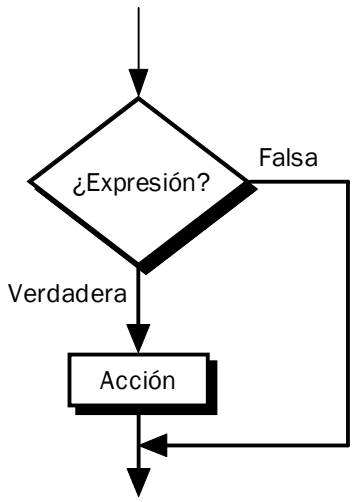



Figura 6.1. Diagrama de flujo de una sentencia básica `if`.

**Ejemplo 6.5.** Prueba de divisibilidad

```
#include <stdio.h>
void main()
{
   int n, d;
```

```
   printf ("Introduzca dos enteros: ");
   scanf ("%d%d", &n,&d);
   if (n%d == 0)
      printf ("%d es divisible por %d\n", n, d);
}
```

**Salida:**

```
Introduzca dos enteros: 36 4
36 es divisible por 4

Introduzca dos enteros: 35 4
```

**Ejemplo 6.6.**

```
#include <stdio.h>
 void main()
{
   float numero;

   // Obtener numero introducido por usuario
   printf ("Introduzca un numero positivo o negativo: ");
   scanf ("%d", &numero);
   // comparar numero con cero
   if (numero > 0)
      printf ("%d es mayor que cero\n", numero);
   if (numero < 0)
      printf ("%d es mayor que cero\n", numero);
   if (numero == 0)
      printf ("%d es igual a cero\n", numero);
}
```

**Salida:**

```
Introduzca un numero positivo o negativo: 10.15
10.15 es mayor que cero
```

## 5.2. Sentencia `if – else`

Esta sentencia permite realizar una *bifurcación*, ejecutando una parte u otra del programa según se cumpla o no una cierta condición. La forma general es la siguiente:

```
if (expresion)
   sentencia1;
else
   sentencia2;
```

En este formato **sentencia1** y **sentencia2** pueden ser sentencias individuales, y por tanto, finalizan con punto y coma, o bien pueden ser grupos de sentencias encerradas entre llaves. Cuando se ejecuta la sentencia `if-else` se evalúa **expresion**. Si expresion es verdadera, se ejecuta la sentencia o grupo de sentencias **sentencia1**; y en caso contrario se ejecuta la sentencia o grupo de sentencias **sentencia2**. La Figura 6.2 muestra la semántica de la sentencia `if-else`.

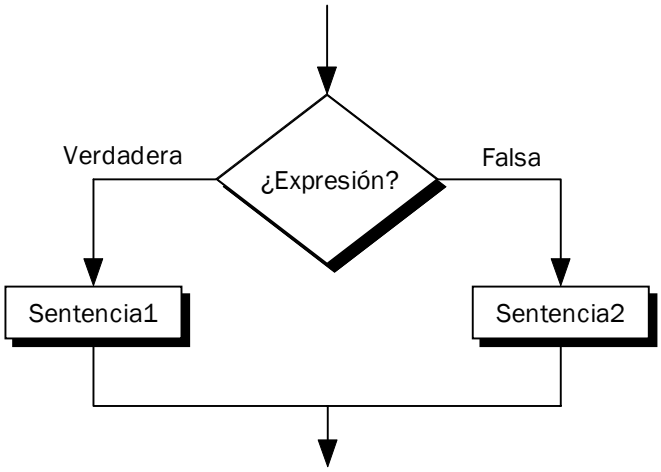



Figura 6.2. Diagrama de flujo de una sentencia if-else.

**Ejemplo 6.7.** Prueba de divisibilidad con if-else

```
#include <stdio.h>
void main()
{
   int n, d;
   printf ("Introduzca dos enteros: ");
   scanf ("%d%d", &n, &d);
   if (n%d == 0)
      printf ("%d es divisible por %d\n", n, d);
   else
      printf ("%d no es divisible por %d\n", n, d);
}
```

**Salida:**

```
Introduzca dos enteros: 35 4
35 no es divisible por 4
```

**Ejemplo 6.8.** El mayor de dos números

```
#include <stdio.h>
void main()
{
   int x, y;
   printf ("Introduzca dos enteros: ");
   scanf (%d%d", &x, &y);
   if ( x > y )
      printf ("%d\n", x);
   else
      printf ("%d\n", y);
}
```

**Salida:**

```
Introduzca dos enteros: 17 54
54
```

## 5.3. Sentencia `if – else` múltiple

Esta sentencia permite realizar una ramificación múltiple, ejecutando *una* entre varias partes del programa según se cumpla *una* entre *n* condiciones. La forma general es la siguiente:

```
if (expresion_1)
   sentencia_1;
else if (expresion_2)
   sentencia_2;
else if (expresion_3)
   sentencia_3;
else if (...)
   ...
[else
sentencia_n;]
```

**Explicación:** Se evalúa **expresion_1**. Si el resultado es ***true***, se ejecuta **sentencia_1**. Si el resultado es ***false***, se salta **sentencia_1** y se evalúa **expresion_2**. Si el resultado es ***true*** se ejecuta **sentencia_2**, mientras que si es ***false*** se evalúa **expresion_3** y así sucesivamente. Si ninguna de las expresiones o condiciones es ***true*** se ejecuta **expresion_n** que es la opción por defecto (puede ser la sentencia vacía, y en ese caso puede eliminarse junto con la palabra **else**). Todas las sentencias pueden ser simples o compuestas.

Ejemplo 6.9. Sentencias compuesta `if-else`

```
// Ilustra las sentencias compuestas if-else
#include <stdio.h>
void main()
{
   int numero;

   printf ("Introduzca un numero positivo o negativo: ");
   scanf ("%d", &numero);

   // comparar numero a numero
   if (numero > 0)
   {
      printf ("%d es mayor que cero.\n", numero);
      printf ("Pruebe de nuevo; introduzca un numero negativo\n");
   }
   else if (numero < 0)
   {
      printf ("%d es menor que cero.\n", numero);
      printf ("Pruebe de nuevo; introduzca un numero negativo\n");
   }
   else
   {
      printf ("%d es igual a cero.\n", numero);
      printf ("Pruebe de nuevo; introduzca un numero negativo\n”);
   }
}
```

**Ejemplo 6.10.** Diferentes formas de escribir sentencias `if` anidadas

**Forma 1:**

```
if (a > 0) if (b > 0) ++a; else if (c > 0)
if (a < 5) ++b; else if (b < 5) ++c; else –a;
else if (c < 5) –b; else –c; else a = 0;
```

**Forma 2:**

```
if (a > 0)
```

```
   if (b > 0)
      ++a;
   else if (c > 0)
      if (a < 5)
         ++b;
      else if (b < 5)
         ++c;
      else -a;
   else if (c < 5)
      -b;
   else
      -c;
else
   a = 0;
```

Forma 3:

```
if (a > 0)
{
   if (b > 0) ++a;
   else if (c > 0)
   {
       if (a < 5) ++b;
       else if (b < 5) ++c;
       else -a;
   }
   else if (c < 5) -b;
   else -c;
}
else
   a = 0;
```

## 5.4. La sentencia `switch`

La sentencia switch es una sentencia C/C++ que se utiliza para seleccionar una de entre múltiples alternativas. Desarrolla una función similar a la de la sentencia `if - else` con múltiples ramificaciones, aunque también importantes diferencias. La forma general de la sentencia `switch` es la siguiente:

```
switch (selector)
{
   case etiqueta_1:
      sentencias_1;
      break;
   case etiqueta_2:
      sentencias_2;
      break;
   ...
   case etiqueta_n:
      sentencias_n;
      break;
   default:
      sentencias;  // opcional
}
```

Se evalúa la expresión de control o *selector*, y se compara el resultado con cada una de las expresiones constantes de `case`. La expresión selector debe ser un valor entero. Cada etiqueta es un valor único, constante, y cada etiqueta debe tener un valor diferente. Si el valor de la expresión *selector* es igual a una de las etiquetas `case` —por ejemplo, *etiqueta_i*— entonces la ejecución comenzará con la primera sentencia de la secuencia *sentencia_i* y continuará hasta que se encuentre una sentencia `break` (o hasta el final de la sentencia de control `switch`). El tipo de cada etiqueta ha de ser el mismo que el de *selector*. En la Figura 6.3.a se muestra el diagrama de flujo general para una sentencia `switch`, mientras que en la Figura 6.3.b se muestra cómo sería el flujo de programa sin las sentencias `break`.

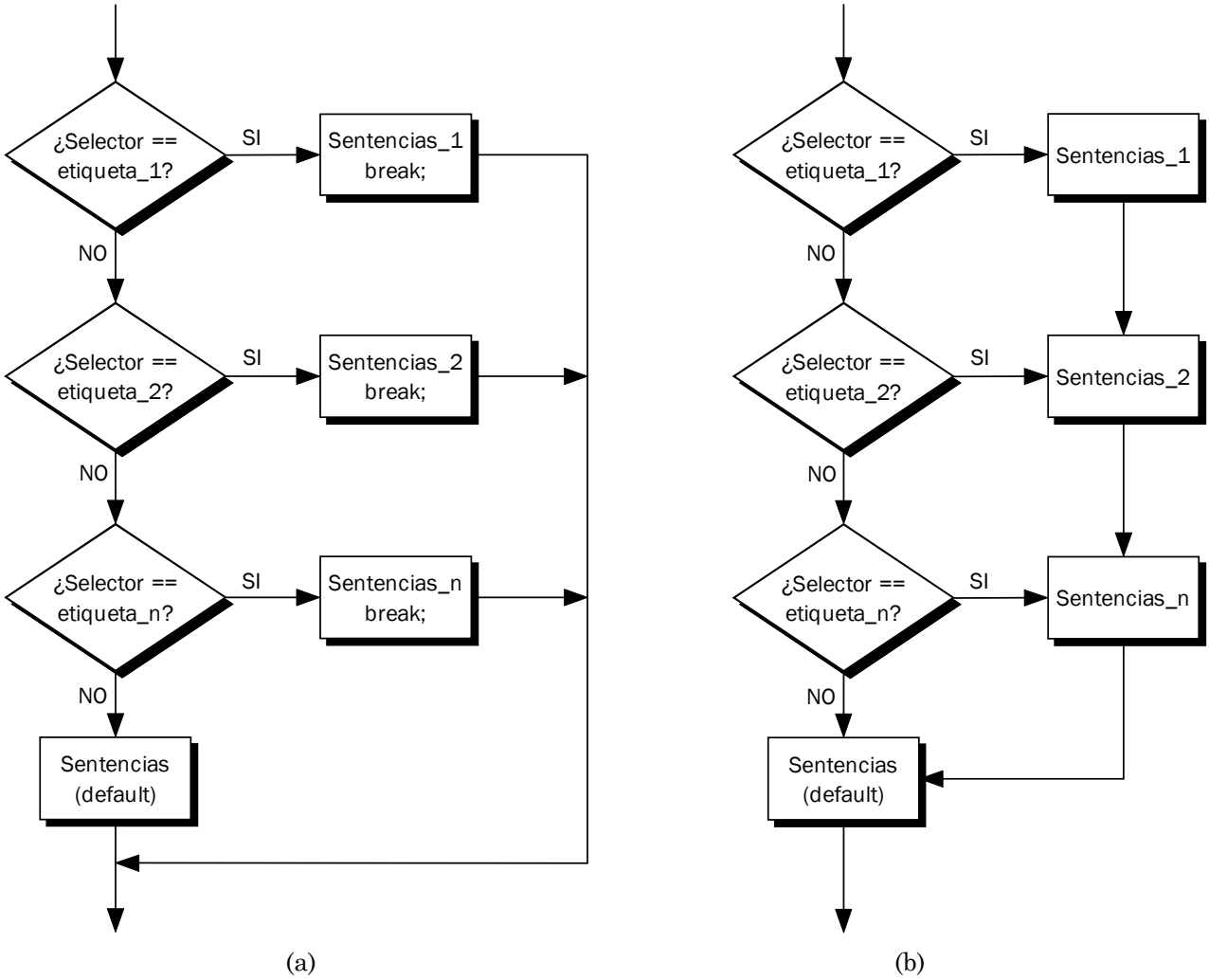



Figura 6.3. Diagrama de flujo de una sentencia `switch`: a) con sentencias `break`; b) sin sentencias `break`.

Si el valor del selector no coincide con ninguna de las etiquetas `case`, no se ejecutará ninguna de las opciones a menos que se especifique una opción por defecto (`default`). Aunque la etiqueta `default` es opcional, su uso es altamente recomendable a menos que se tenga la completa seguridad de que todos los valores de selector están incluidos en las etiquetas case.

### La sentencia `break`

Esta sentencia consta de la palabra reservada `break` seguida por un punto y coma. La ejecución de una sentencia `switch` finaliza cuando el procesador encuentra una sentencia `break`.

Ejemplos de utilización de `break`

```
switch (opcion)
{
   case 0:
      printf ("Cero!\n");
      break;
   case 1:
      printf ("Uno!\n");
      break;
   case 2:
      printf ("Dos!\n");
      break;
```

```
  default:
      printf ("Fuera de rango\n");
}

...

switch (opcion)
{
   case 0:
   case 1:
      printf ("Menor que 3");
      break;
   case 3:
      printf ("Igual a 3");
      break;
   default:
      printf ("Mayor que 3");
}
```

En el siguiente ejemplo se muestra el código para determinar si un determinado carácter es una vocal, utilizando primero sentencias `if-else`, y después una sentencia `switch`.

**Ejemplo 6.11.** Comparación de sentencias

```
...
// solucion con if-else
if ( (car == 'a') || (car == 'A' ))
   printf ("%c es una vocal\n", car);
else if ( (car == 'e') || (car == 'E' ))
   printf ("%c es una vocal\n", car);
else if ( (car == 'i') || (car == 'I' ))
   printf ("%c es una vocal\n", car);
else if ( (car == 'o') || (car == 'O' ))
   printf ("%c es una vocal\n", car);
else if ( (car == 'u') || (car == 'U' ))
   printf ("%c es una vocal\n", car);
else
   printf ("%c no es una vocal\n", car);
...

// Solución con switch
switch(car)
{
   case 'a': case 'A':
   case 'e': case 'E':
   case 'i': case 'I':
   case 'o': case 'O':
   case 'u': case 'U':
      printf ("%c es una vocal\n", car);
      break;
   default:
      printf ("%c no es una vocal\n", car);
}
...
```

**Ejemplo 6.12.** Programa de ilustración de `switch`

```
// Programa de ilustracion de la sentencia switch
#include <stdio.h>

int main()
{
```

```
   char nota;
   printf ("Introduzca calificacion (A-H) y pulse Intro: ");
   scanf ("%c", &nota);

   switch (nota)
   {
      case 'A': printf ("Excelente. Examen superado\n");
                break;
      case 'B': printf ("Notable. Suficiencia \n");
                break;
      case 'C': printf ("Aprobado\n");
                break;
      case 'D':
      case 'F': printf ("Suspenso\n");
                break;
      default:  printf ("No es posible esta nota\n");
   }
   printf ("Final del programa\n");
   return 0;
}
```

**Salidas:**

```
Introduzca calificacion (A-H) y pulse Intro: A
Excelente.Examen superado
Final del programa
```

```
Introduzca calificacion (A-H) y pulse Intro: e
No es posible esta nota
Final del programa
```

**Ejemplo 6.13.** Uso de la sentencia `switch` en menús

```
// Programa de ilustracion de la sentencia switch
#include <stdio.h>    // para operaciones de E/S

int main()
{
   int tipo_vehiculo;
   float peaje;

   printf("1. Turismo.\n");
   printf("2. Autobus.\n");
   printf("3. Motocicleta.\n");
   printf("Seleccione el numero correspondiente a su vehiculo\n";
   printf("y pulse Intro: ");
   scanf("%d", &tipo_vehiculo);

   switch (tipo_vehiculo)
   {
      case 1: // Turismo
         peaje = 500.;
         printf("La tarifa de un turismo es de %.0f ptas. ", peaje);
         printf("( %.2f euros)\n", peaje/166.386);
         break;
      case 2: // Autobus
         peaje = 3000.;
         printf("La tarifa de un autobus es de %.0f ptas. ", peaje);
         printf("( %.2f euros)\n", peaje/166.386);
         break;
      case 3: // Motocicleta
         peaje = 300.;
           printf("La tarifa de una motocicleta es de %.0f ptas. ", pea-
je);
```

```
         printf("( %.2f euros)\n", peaje/166.386);
         break;
      default:  printf("Vehiculo no autorizado\n");
   }
   printf("Final del programa\n");
   return 0;
}
```

**Salidas:**

```
1. Turismo.
2. Autobus.
3. Motocicleta.
Seleccione el numero correspondiente a su vehiculo
 y pulse Intro: 1
La tarifa de un turismo es de 500 ptas. ( 3.01 euros)
Final del programa
```

```
1. Turismo.
2. Autobus.
3. Motocicleta.
Seleccione el numero correspondiente a su vehiculo
 y pulse Intro: 5
Vehiculo no autorizado
Final del programa
```

## 5.5. Sentencias `if` anidadas

Una sentencia `if` puede incluir otros `if` dentro de la parte correspondiente a su sentencia, A estas sentencias se les llama **sentencias anidadas** (una dentro de otra). Por ejemplo:

```
if (a >= b)
   if (b != 0.0)
      c = a/b;
```

En ocasiones pueden aparecer dificultades de interpretación con sentencias `if...else`anidadas, como en el caso siguiente:

| Formato 1 | Formato 2 |
|---|---|
| `if (a >= b)`<br>`  if (b != 0.0)`<br>`  c = a/b;`<br>`else`<br>`  c = 0.0;` | `if (a >= b)`<br>`  if (b != 0.0)`<br>`  c = a/b;`<br>`else`<br>`  c = 0.0;` |

En principio se podría plantear la duda de a cuál de los dos `if` corresponde la parte `else` del programa. Los espacios en blanco del formato 1 —las indentaciones de las líneas— parecen indicar que la sentencia `else` corresponde al segundo de los `if`, mientras que en el formato 2 parece corresponder al primer `if`. La respuesta correcta es la que indica el formato 1, pues la regla es que el `else` pertenece al `if` más cercano. Sin embargo, no hay que olvidar que el compilador no considera los espacios en blanco (aunque sea muy conveniente introducirlos para hacer más claro y legible el programa), y que si se quisiera que el `else` perteneciera al primero de los `if` no bastaría cambiar los espacios en blanco, sino que habría que utilizar llaves, en la forma:

```
if (a >= b)
{
   if (b != 0.0)
      c = a/b;
}
```

```
else
   c = 0.0;
```

# 6. Estructuras repetitivas

Una de las características de los ordenadores que aumentan considerablemente su potencia es la capacidad para ejecutar una tarea muchas veces con gran velocidad, fiabilidad y precisión. En este tema se estudian las estructuras de control iterativas o repetitivas, que, como su nombre indica, permiten realizar la iteración o repetición de acciones. C++ soporta tres tipos de estructuras de control: los *bucles* `for`, `while`, y `do-while`. Este tipo de estructuras permiten ejecutar más de una vez un mismo bloque de sentencias.

La sentencia o grupo de sentencias que se repiten se denomina **cuerpo** del bucle y cada repetición del cuerpo del bucle se llama **iteración** del bucle.

## 6.1. La sentencia `for`

La sentencia `for` es un método para ejecutar un bloque de sentencias un número definido de veces. La sintaxis es la siguiente:

```
for (Inicialización; Condición; Actualización)
Sentencias
```

El bucle `for` contiene las siguientes partes:

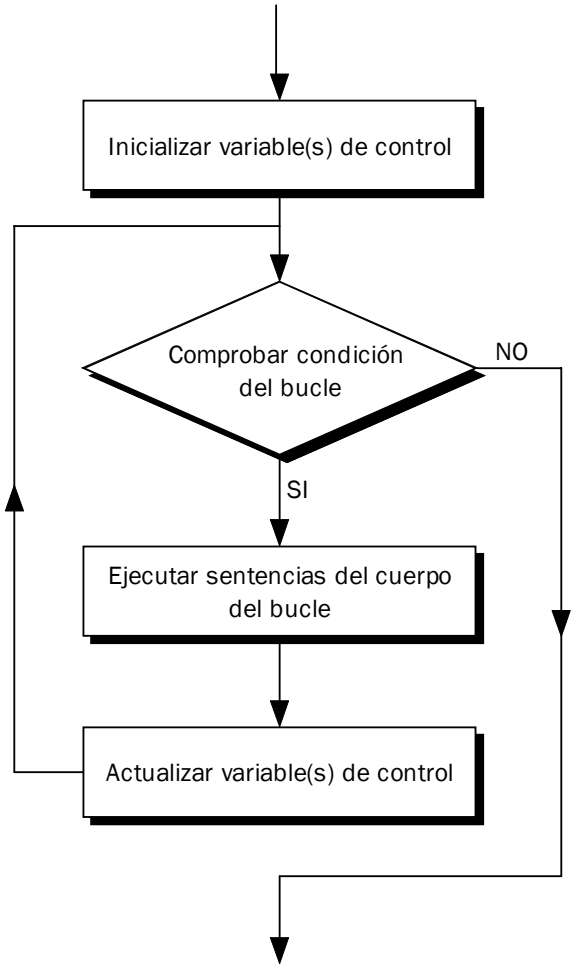
- **Inicialización**: inicializa las variables de control del bucle. Se pueden utilizar varias variables de control
- **Condición**: expresión lógica que determina si las sentencias del bucle se han de realizar (condición cierta, `true`) una vez más o si el bucle debe finalizar (condición falsa, `false`)
- **Actualización**: permite modificar el valor de las variables de control del bucle (por lo general se incrementa o decrementa su valor).
- **Sentencias**: acciones que se ejecutarán en cada iteración del bucle

**Ejemplos:**

```
// Imprimir Hola 10 veces
for (int i=0; i < 10; i++)
   printf ("Hola!");
```

```
// Imprimir Hola 10 veces
for (int i=0; i < 10; i++)
{
   printf ("Hola!");
   printf ("El valor de i es: %d\n",
i);
}
```

Veamos el siguiente esquema para comprender mejor el funcionamiento de esta estructura:




Existen dos formas de implantar la sentencia `for`, que se utilizan normalmente como bucles contadores: formato *ascendente*, en el que la variable de control se incrementa, y formato *descendente*, en el que la variable de control se decrementa.

| | |
|---|---|
| `for(int var_control = valor_inicial; var_control <= valor_limite; exp_incremento)`<br>`  sentencia;` | `for(int var_control = valor_inicial; var_control >= valor_limite; exp_decremento)`<br>`  sentencia;` |
| `for (int n = 1; n <= 5; n++)`<br>`  printf ("%d\t%dn" n, n*n);` | `for (int n = 10; n >= 5; n--)`<br>`  printf ("%d\t%dn" n, n*n);` |
| `1  1`<br>`2  4`<br>`3  9`<br>`4  16`<br>`5  25` | `10  100`<br>`9  81`<br>`8  64`<br>`7  49`<br>`6  36`<br>`5  25` |

## 6.2. Casos particulares del bucle `for`

Un bucle `for` debe construirse con gran precaución, de forma que se asegure que el programa salga del bucle en algún momento, esto es, que la condición deje de cumplirse en alguna iteración. También es po-

sible modificar los valores de las variables de control desde las sentencias del cuerpo del bucle; incluso se puede utilizar la sentencia `break`.

Veamos a continuación algunos casos particulares que pueden darse en un bucle `for`.

### 6.2.1. Bucles infinitos

El uso de un bucle `for` es implantar bucles contadores, en los que se conoce de antemano el número exacto de iteraciones. Sin embargo, existen muchos problemas en los que el número de iteraciones no se conoce a priori. En estos casos se puede implantar un **bucle infinito**. Su sintaxis es la siguiente:

```
for ( ; ; )
{
   sentencias
}
```

El bucle se ejecutará indefinidamente a menos que se utilice una sentencia `break`. La razón de que el bucle se ejecute indefinidamente es que se han omitido los tres campos (inicialización, condición y actualización), y, por tanto, no hay una condición que especifique el fin del bucle.

Para evitar esta situación, el bucle `for` ha de contener una sentencia que termine la ejecución del programa cuando se cumpla una determinada condición. Esta sentencia suele ser del tipo

```
if (condicion)   break;
```

**Ejemplo:**

```
...
for ( ; ; )
{
   printf << "Introduzca un numero: ";
   scanf ("%d", &num);
   if (num = -999) break;
   ...
}
...
```

### 6.2.2. Bucles `for` vacíos

Un error bastante frecuente es colocar un punto y coma después de la cabecera del bucle `for`:

```
for (int i = 0, i <= 10; i++) ;
   printf ("Hola Mundo!\n");
```

Este trozo de código no da como resultado que *"Hola Mundo!"* aparezca 11 veces en la pantalla. En realidad solamente aparecerá una vez, ya que el bucle `for` acaba en el punto y coma, y, por tanto, no tiene cuerpo (o está *vacío*). El bucle no hace nada, y después se ejecuta la sentencia `cout ...` que no pertenece al bucle.

El bucle `for` vacío puede utilizarse en algunas aplicaciones como temporizador.

## 6.3. Más ejemplos del bucle `for`

**Ejemplo 6.14**: Programa que visualiza la tabla ASCII.

```
#include <stdio.h>
void main()
{
   int i ;
   for(i=1 ;i <= 255 ; i++)
```

```
      printf ("%d = %c \t, i, (char)i);
}
```

Observa la línea `printf`, la misma variable i sirve para visualizar el número entero que contiene, y su código ASCII correspondiente: `i` muestra el número, y `char(i)` muestra el carácter ASCII que este número representa.

**Ejemplo 6.15:** Programa que muestra los cien primeros números negativos. Observa en este programa que en lugar de incrementar la variable de control, esta se decrementa.

```
#include <stdio.h>
void main()
{
   int i ;
   for (i=-1 ;i >= -100 ; i--)
      printf ("%d", i);
}
```

**Ejemplo 6.16:** Programa que visualiza los números comprendidos entre 50 y 100 en intervalos de 4. Observa que el valor inicial para la variable de control es 50, y que se incrementa de 4 en 4.

```
#include <stdio.h>
void main()
{
   int i ;
   for ( i = 50 ; i<= 100 ; i=i+4)
         printf ("%d\t", i);
}
```

**Ejemplo 6.17**: Programa que imprime los 100 primeros números positivos y los 100 primeros números negativos de la siguiente manera: 1, -1, 2, -2, 3, -3, 4, -5,....., 99, -99, 100, -100. Observa que en la parte de iniciación del bucle, se asignan valores a dos variables, y en la parte de incremento, también se incrementan 2.

```
#include <stdio.h>
void main()
{
   int i, j;
   for(i=1 , j=-1 ;i <= 100 ; i++, j--)
      printf ("%d, %d, ", i, j);
}
```

En la parte de iniciación del bucle, se puede dar valores a tantas variables como se quiera; cada asignación ha de ir separada por comas. En la parte de incremento, se puede cambiar el valor a las variables que se desee; cada asignación se ha de separar por comas. Solo se puede establecer una única condición de finalización de bucle y siempre separada por punto y coma de las expresiones iniciales y de incremento.

En un bucle for se puede prescindir de las expresiones de inicialización de bucle, y de la expresión de incremento. Veamos los ejemplos siguientes:

**Ejemplo 6.18**: Programa que muestra los valores entre un número introducido por el teclado, y visualiza los valores entre este valor y 10. Si el número entrado es > 10 no se visualiza nada.

```
#include <stdio.h>
void main()
{
   int i ;
```

```
   printf ("Introduzca el valor inicial : ");
   scanf ("%d", &i);
   for ( ;i<=10 ;i++)
      printf ("%d\n", i);
}
```

**Ejemplo 6.19:** Programa que muestra números del 1 al 10 con incrementos introducidos por el teclado.

```
#include <stdio.h>
void main()
{
   int i, incremento ;
   for(i=1 ;i<=10 ;)
   {
      printf ("%d\n", i);
      printf ("Incremento?: ");
      scanf ("%d", &incremento);
      i = i + incremento ;
   }
}
```

**Ejemplo 6.20:** Programa que introduce valores, hasta que se introduzca uno mayor que 10.

```
#include <stdio.h>
void main()
{
   int i = 0; // Iniciada para que sea <= 10.
   for(;i<=10;)
   {
      printf ("Introduce un valor: ");
      scanf ("%d", &i);
   }
}
```

Las tres estructuras anteriores son realmente rebuscadas, raramente se utilizan; por otra parte se hubiese podido resolver más "naturalmente" con otro tipo de bucle.

## 6.4. Bucles `for` anidados

De la misma manera que vimos en el caso de las estructuras selectivas, un bucle `for` puede contener a su vez otros bucles `for`.

**Ejemplo 6.21:** Programa que muestra todos los resultados posibles que pueden salir al tirar dos dados. Combinaremos el resultado de un dado con los seis que pueden salir del otro.

```
#include <stdio.h>
void main()
{
   int dado2;
   for ( dado2=1; dado2 <=6; dado2++)
      printf (" 1, %d", dado2);
   for ( dado2=1 ; dado2 <=6 ; dado2++)
      printf (" 2, %d", dado2);
   for ( dado2=1 ; dado2 <=6 ; dado2++)
      printf (" 3, %d", dado2);
   for ( dado2=1 ; dado2 <=6 ; dado2++)
      printf (" 4, %d" << dado2);
   for ( dado2=1 ; dado2 <=6 ; dado2++)
      printf (" 5, %d", dado2);
   for ( dado2=1 ; dado2 <=6 ; dado2++)
```

```
      printf (" 6, %d", dado2);
}
```

En este programa nos encontramos con el mismo caso que con la misma versión del programa de las notas (la que no utilizaba el bucle ***for***): hay una sentencia que se repite 6 veces (la única diferencia es el valor que representa el primer dado, valor del 1 al 6). Cerraremos esta sentencia dentro de un bucle ***for***, y el programa quedará de la siguiente manera:

```
#include <stdio.h>
void main()
{
   int dado1, dado2 ;
   for (dado1=1; dado1 <= 6; dado1++)
   {
      for (dado2 = 1; dado2 <=6; dado2++)
         printf ("\t%d , %d", dado1, dado2) ;
      printf ("\n");
   }
}
```

**Salida:**

```
        1,1     1,2     1,3     1,4     1,5     1,6
        2,1     2,2     2,3     2,4     2,5     2,6
        3,1     3,2     3,3     3,4     3,5     3,6
        4,1     4,2     4,3     4,4     4,5     4,6
        5,1     5,2     5,3     5,4     5,5     5,6
        6,1     6,2     6,3     6,4     6,5     6,6
```

**Ejemplo 6.22:** Programa que visualiza la cantidad de asteriscos que representa cada línea de la pantalla

```
#include <stdio.h>
void main()
{
   int numlinea, numasteriscos ;
   for (numlinea = 1 ; numlinea <= 20 ; numlinea++)
   {
      for ( numasteriscos = 1 ; numasteriscos <= numlinea ; numasteris-
cos++)
         printf ("*");
      printf ("\n");      // Para hacer salto de línea
   }
}
```

## 6.5. La sentencia `while`

Supongamos que tenemos que hacer un programa que sume números que se van introduciendo por teclado, pero que en principio, no se sabe la cantidad de números que se van a entrar.

Utilizando el bucle for, que ya conocemos, podríamos llegar a solucionarlo de la manera siguiente:

```
#include <stdio.h>
void main()
{
   int suma = 0;
   int numero = 1 ;       // para entrar al bucle
   for ( ; numero > 0 ;)
   {
```

```
      printf ("Introduce un numero (0 para acabar la serie) : ");
      scanf ("%d", &numero);
      suma = suma + numero ;
   }
   printf ("La suma de los valores introducidos es : %d\n", suma);
}
```

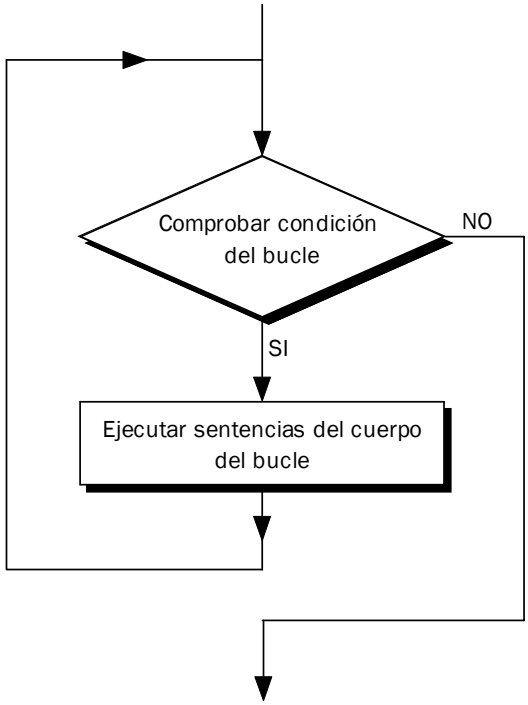
Es una forma rebuscada, poco intuitiva, y nada elegante. La mayoría de los lenguajes de programación de alto nivel (por no decir todos), tienen una estructura repetitiva para controlar acciones repetitivas que en principio, no se sabe cuantas veces se ha de ejecutar. En C/C++ esta estructura es `while`, y la repetición de las sentencias esta controlada por una o más condiciones.

### 6.5.1. Estructura general del bucle `while`

La sintaxis del bucle `while` es la siguiente:

| Cuerpo con varias sentencias | Cuerpo con una sentencia |
|---|---|
| `while(condicion)`<br>`{`<br>`  sentencias;`<br>`}` | `while(condicion)`<br>`  sentencia;` |

En la siguiente figura se muestra el diagrama de flujo de la sentencia while.




Cuando el programa llega a la sentencia `while`:

1. Comprueba si se cumple la condición (de la misma manera que un `if`)

2. Si la condición se cumple, pasa a ejecutar las sentencias del cuerpo de bucle, y se vuelve al paso 1; si no se cumple la condición, pasa a la línea siguiente fuera del bucle.

El programa anterior implantado con una sentencia while sería:

```
#include <stdio.h>
```

```
void main()
{
   int suma = 0;
   int numero = 1 ;      // para entrar al bucle
   printf ("Introduce un numero (0 para acabar la serie) : ");
   scanf ("%d", &numero);

   while (numero !=0)
   {
      suma = suma + numero ;
      printf ("Introduce un numero (0 para acabar la serie) : ");
      scanf ("%d", &numero);
   }
   printf ("La suma de los valores introducidos es : %d\n", suma);
}
```

En resumen, el cuerpo del bucle while se repite mientras la condición (expresión lógica) sea cierta. Cuando la condición no se cumple (false), el bucle termina y se ejecuta la siguiente sentencia del programa

### 6.5.2. Casos particulares con bucles while

#### Terminaciones anormales de un ciclo

Un error típico en el diseño de una sentencia while se produce cuando el cuerpo del bucle se codifica como una sola sentencia, cuando en realidad tiene varias sentencias. El código siguiente:

```
contador = 1;
while (contador < 25)
   printf ("%d\n", contador);
   contador ++;
```

visualizará infinitas veces el valor 1. El programa entra en un bucle infinito del que nunca sale porque no se actualiza la variable de control contador, aunque aparentemente el sangrado puede dar la sensación de que el cuerpo de while tiene dos sentencias (`cout...` y `contador ++;` ). La solución es inmediata: utilizar las llaves.

```
contador = 1;
while (contador < 25)
{
printf ("%d\n", contador);
   contador ++;
}
```

#### Bucle while con cero iteraciones

El cuerpo de un bucle no se ejecuta nunca si la condición del bucle no se cumple cuando se alcanza el while por primera vez.

```
contador = 10;
while (contador > 100)
{
   ...
}
```

#### Bucle controlado por centinela

El centinela es un valor que sirve para terminar el proceso del bucle. El usuario debe introducir como último dato el valor centinela. La condición del bucle comprueba cada dato y termina cuando se lee el valor centinela.

Ejemplo **6.23**: bucle con centinela

```
// Entrada de datos numericos
// El centinela es -1
```

```
#include <stdio.h>
void main()
{
   int nota, cuenta, suma;
   int centinela = -1;
   printf ("Introduzca primera nota: ");
   scanf ("%d", &nota;
   while (nota != centinela)
   {
      cuenta ++;
      suma += nota;
      printf ("Introduzca la siguiente nota: ");
      scanf ("%d", &nota);
   } // fin del bucle
   printf ("Final");
}
```

**Salida:**

```
Introduzca primera nota: 5
Introduzca la siguiente nota: 3
Introduzca la siguiente nota: 8
Introduzca la siguiente nota: -1
```

### Bucles while (true)

Se puede crear un bucle while infinito utilizando el valor true como condición.

Ejemplo **6.24**. Bucle while infinito

```
#include <stdio.h>
void main()
{
   int contador = 0;
   while (1) // tambien while(true)
   {
      contador++;
      if(contador > 10)
         break;
   }
   printf ("Contador: %d\n", contador);
}
```

**Salida:**

```
contador: 11
```

### Bucles `while` anidados

De la misma manera que se encadenan bucles `for`, también se pueden encadenar bucles `while`. Vea el ejemplo siguiente

Ejemplo **6.25**. Programa para sumar diferentes series de números. El programa acaba cuando el usuario introduce un 0 como cantidad de términos para la serie.

```
// Bucles while anidados
#include <stdio.h>
void main()
{
   int NumTerminos, Termino, Suma, Contador;

   printf ("Introduzca la cantidad de terminos para a la serie: ");
```

```
   scanf ("%d", &NumTerminos);
   while (NumTerminos > 0)
   {
      Contador = 0;
      Suma = 0;
      while (Contador < NumTerminos)
      {
         printf ("Entre termino %d: ", Contador);
         scanf ("%d", &Termino);
         Suma = Suma + Termino; // tambien suma+=termino
         Contador = Contador + 1;
      }
      printf ("El valor de la suma es: %d\n", Suma);
      printf ("Introduzca la cantidad de terminos para la serie: ");
      scanf ("%d", NumTerminos);
   }
}
```

## 6.6. La sentencia `do-while`

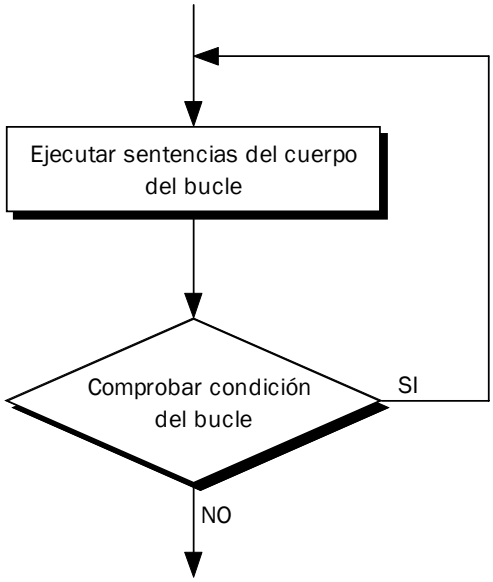
Esta estructura es muy parecida a la estructura *while*; la única diferencia es que en el bucle `do … while` la evaluación de la condición se hace al final, es decir, después de haber ejecutado el cuerpo del bucle al menos una vez. Observa el programa de introducir números hasta pulsar 0, y compara las dos formas de hacerlo:

En estructuras repetitivas que requieren la ejecución de les instrucciones del cuerpo de bucle al menos una vez, es mejor utilizar la estructura `do … while`

### *6.6.1.* Estructura general del bucle `do-while`

| Cuerpo con varias sentencias | Cuerpo con una sentencia |
|---|---|
| `do`<br>`{`<br>`  Sentencia 1;`<br>`  Sentencia 2;`<br>`  .`<br>`  .`<br>`  Sentencia n;`<br>`} while (Condicion) ;` | `do`<br>`{`<br>`  Sentencia ;`<br>`} while (Condición) ;` |

El bucle `do...while` implantado con lenguaje C/C++, siempre lleva llaves de apertura y cierre. Observa también que la línea `while` acaba siempre con un punto y coma.




Ejemplo 6.26. El programa introduce números hasta pulsar 0 con el bucle ***do..while***.

```
#include <stdio.h>
void main()
{
   int Numero, Suma ;
   do
   {
      printf ("Entre un numero: ");
      scanf ("%d" &Numero);
      Suma = Suma + Numero;
   }
   while (Numero != 0) ;
}
```

### 6.7. Cuando utilizar cada estructura

- Utiliza el bucle `for` cuando sepas a priori, el número de veces que se ha de repetir un proceso.
- Utiliza el bucle `while` cuando no sepas a priori, cuantas veces se ha de repetir un proceso.
- Utiliza el bucle `do..while` cuando no sepas a priori, cuantas veces se ha de repetir un proceso, pero que éste, al menos, se ha de ejecutar una vez.
- Con el bucle `while` se pueden implementar todas las estructuras repetitivas. Hay lenguajes que solamente tienen esta estructura repetitiva.

## 7. Conclusiones

En este tema hemos visto cual son las sentencias básicas de control de la programación estructurada, así como los operadores que se utilizan para la construcción de expresiones condicionales.

Con este tema finaliza el bloque I, y el alumno ya está preparado para implantar pequeñas aplicaciones en C++. En el siguiente bloque se continuará con los conceptos necesarios para comenzar el análisis, diseño e implantación empleando técnicas de desarrollo orientado a procesos.

> ***Nota del profesor****: si detectas algún error, o hay alguna parte confusa, o piensas que sobra o falta alguna sección, por favor, enviadme un mensaje a josdie@tel.uva.es. Gracias por anticipado. Vuestras sugerencias se utilizarán para mejorar estos apuntes.*