Índice general

TEMA 2. ESTRUCTURAS DE CONTROL

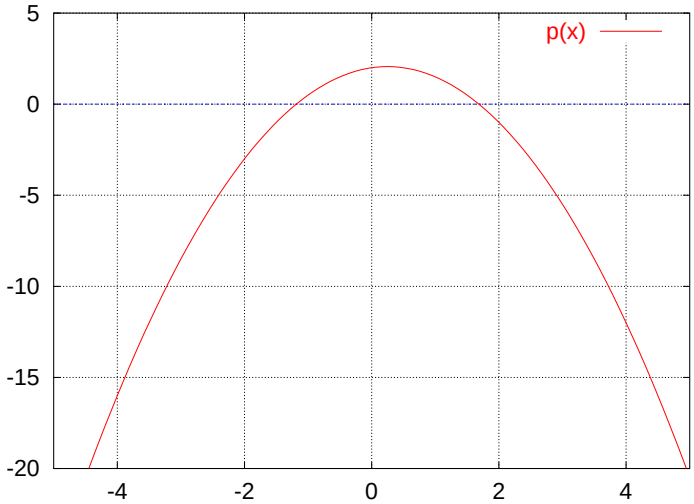
2.1. INTRODUCCIÓN

Estructuras
- **Secuencial**
- **Condicional**
- **Repetitiva**

Estructura Secuencial: las instrucciones se van ejecutando sucesivamente, siguiendo el orden de aparición de éstas. No hay *saltos*.

Ejemplo: *Calcular las raíces de una ecuación de 2º grado*

$$p(x) = ax^2 + bx + c$$



$$r1 = \frac{-b+\sqrt{b^2-4ac}}{2a} \qquad r2 = \frac{-b-\sqrt{b^2-4ac}}{2a}$$

```
/* Programa que calcula las raíces de una ecuación
   de segundo grado  (PRIMERA APROXIMACIÓN)
   Entradas: Los parámetros de la ecuación
   Salidas: Las raíces de la parábola
*/
#include <iostream>
#include <cmath>
using namespace std;

int main(){
  double a,b,c;    // Parámetros de la ecuación
  double r1,r2;    // Raíces obtenidas

1  cout << "\nIntroduce coeficiente de 2º grado: ";
2  cin >> a;
3  cout << "\nIntroduce coeficiente de 1er grado: ";
4  cin >> b;
5  cout << "\nIntroduce coeficiente independiente: ";
6  cin >> c;
7  r1 = ( -b + sqrt( b*b-4*a*c ) ) / (2*a);
8  r2 = ( -b - sqrt( b*b-4*a*c ) ) / (2*a);
9  cout << "Las raíces son" << r1 << " y "
       << r2 << endl;
}
```

Flujo de control: (1,2,3,4,5,6,7,8,9)

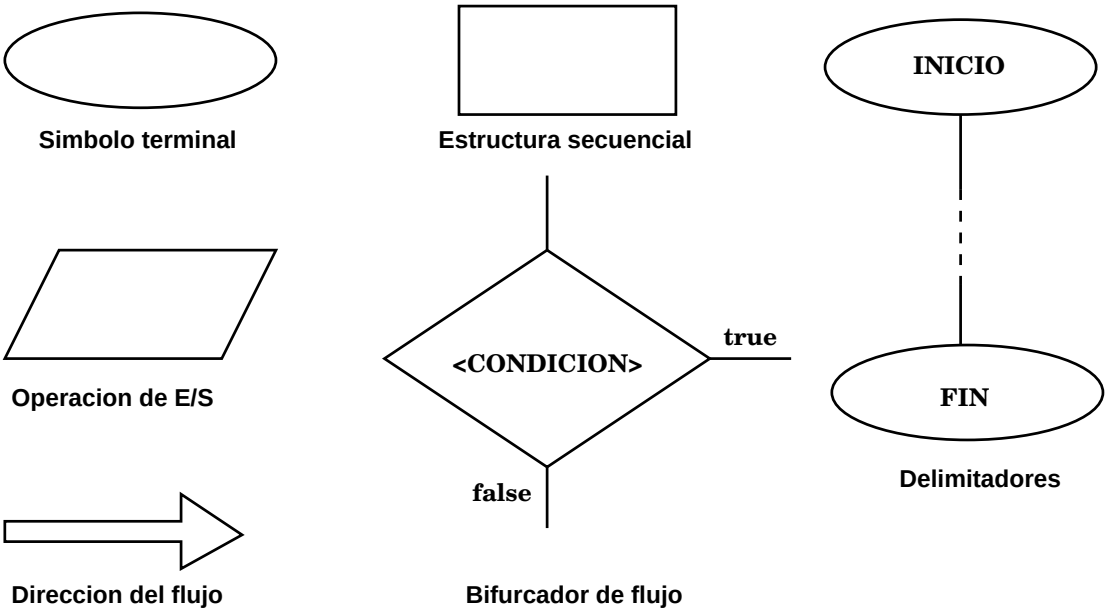
Pregunta: ¿Es `a` distinto de cero?

- **Si. Entonces sigue la ejecución.**
- **No. Entonces salta las sentencias 7–9 y termina.**

2.1.1. UNA NOTACIÓN PARA DESCRIBIR ALGORITMOS

Diagramas de flujo

Símbolos básicos



Símbolos adicionales

Fig. 4.9(a) Subprograma

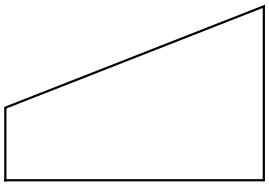


Fig. 4.9(b) Entrada por Teclado

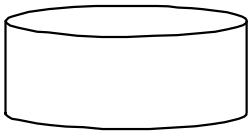


Fig. 4.9(c) Disco Magnetico

Fig. 4.9(d) Impresora

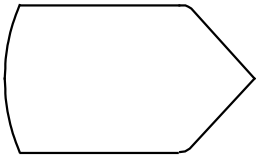


Fig. 4.9(e) Pantalla

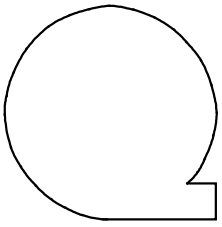
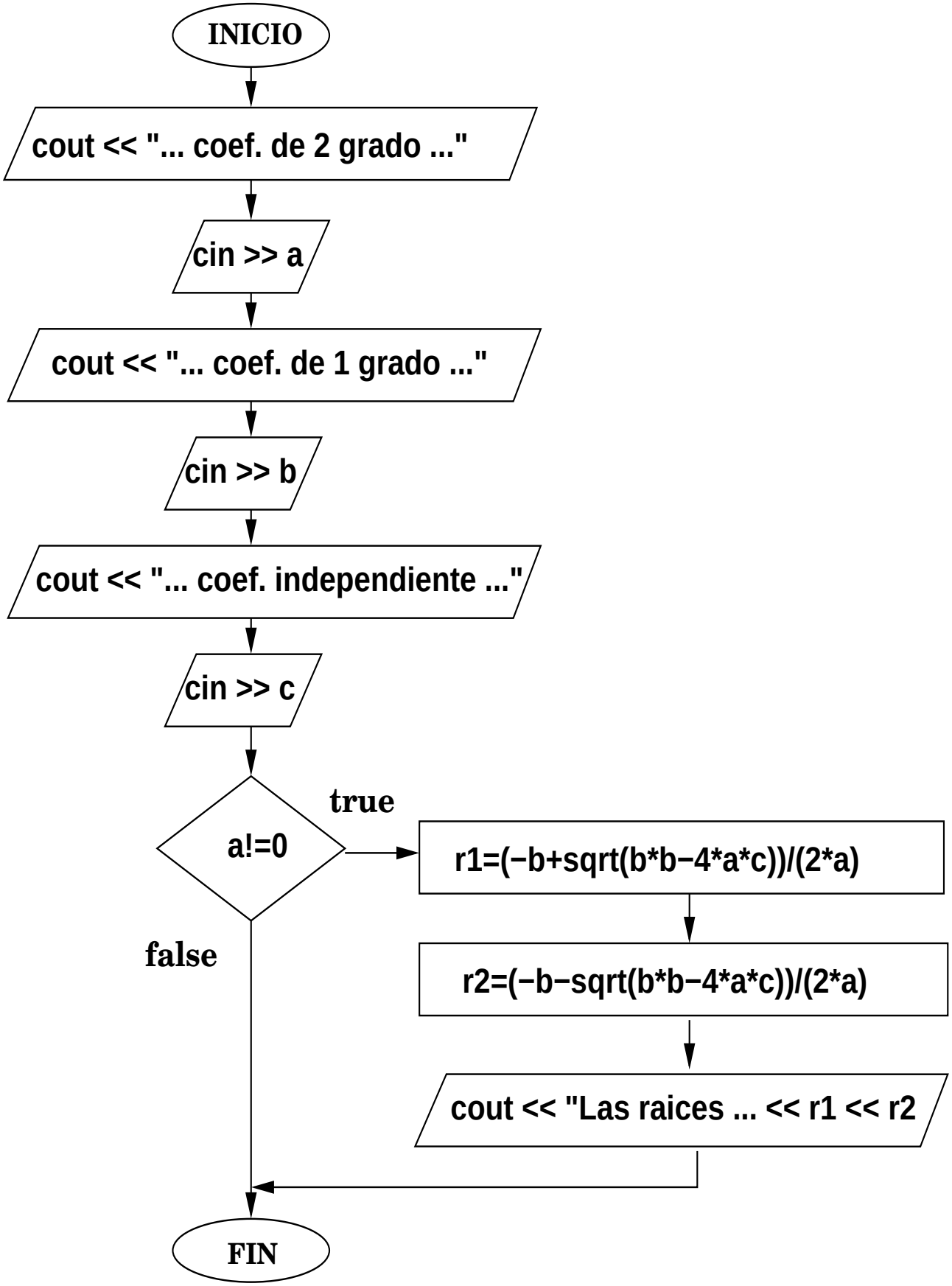


Fig. 4.9(f) Cinta Magnetica

Calcular las raíces de una ecuación de segundo grado



2.2. Estructura Condicional

2.2.1. Introducción

Una *ejecución condicional* consiste en la ejecución de una (o más) sentencia(s) dependiendo de la evaluación de una condición.

Una *estructura condicional* es una estructura que permite una ejecución condicional. Existen tres tipos:

- ***Simple*,**
- ***Doble* y**
- ***Múltiple*.**

2.2.2. Estructura Condicional Simple

```
if (<condición>)
   <bloque if>
```

<*condición*> es una expresión lógica.

<*bloque if*> es el bloque de ejecución condicional

- **Si hay varias sentencias, es necesario encerrarlas entre llaves.**
- **Si sólo hay una sentencia, no son necesarias las llaves.**

```
/* Programa que calcula las raíces de una ecuación
   de segundo grado  (SEGUNDA APROXIMACIÓN)
   Entradas: Los parámetros de la ecuación
   Salidas: Las raíces de la parábola
*/

 #include <iostream>
 #include <cmath>
 using namespace std;

 int main(){
   double a,b,c;    // Parámetros de la ecuación
   double r1,r2;    // Raíces obtenidas

 1  cout << "\nIntroduce coeficiente de 2º grado: ";
 2  cin >> a;
 3  cout << "\nIntroduce coeficiente de 1er grado: ";
 4  cin >> b;
 5  cout << "\nIntroduce coeficiente independiente: ";
 6  cin >> c;

 7  if (a!=0) {
 8     r1 = ( -b + sqrt( b*b-4*a*c ) ) / (2*a);
 9     r2 = ( -b - sqrt( b*b-4*a*c ) ) / (2*a);
10     cout << "Las raíces son" << r1 << " y "
            << r2 << endl;
   }
 }
```

¿Qué pasa si $a = 0$ en la ecuación de segundo grado?
El algoritmo debe devolver $-c/b$

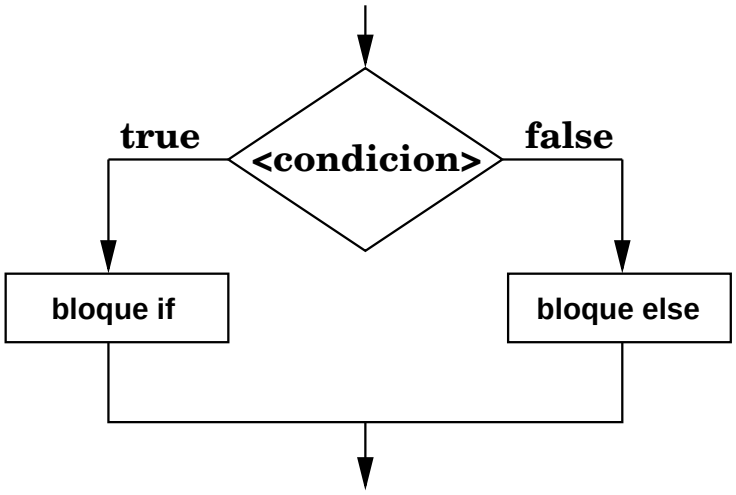
```
if (a!=0) {
   r1 = ( -b + sqrt( b*b-4*a*c ) ) / (2*a);
   r2 = ( -b - sqrt( b*b-4*a*c ) ) / (2*a);
   cout << "Las raíces son " << r1 << " y "
        << r2 << endl;
}

if (a==0)
   cout << "Tiene una única raíz" << -c/b << endl;
```

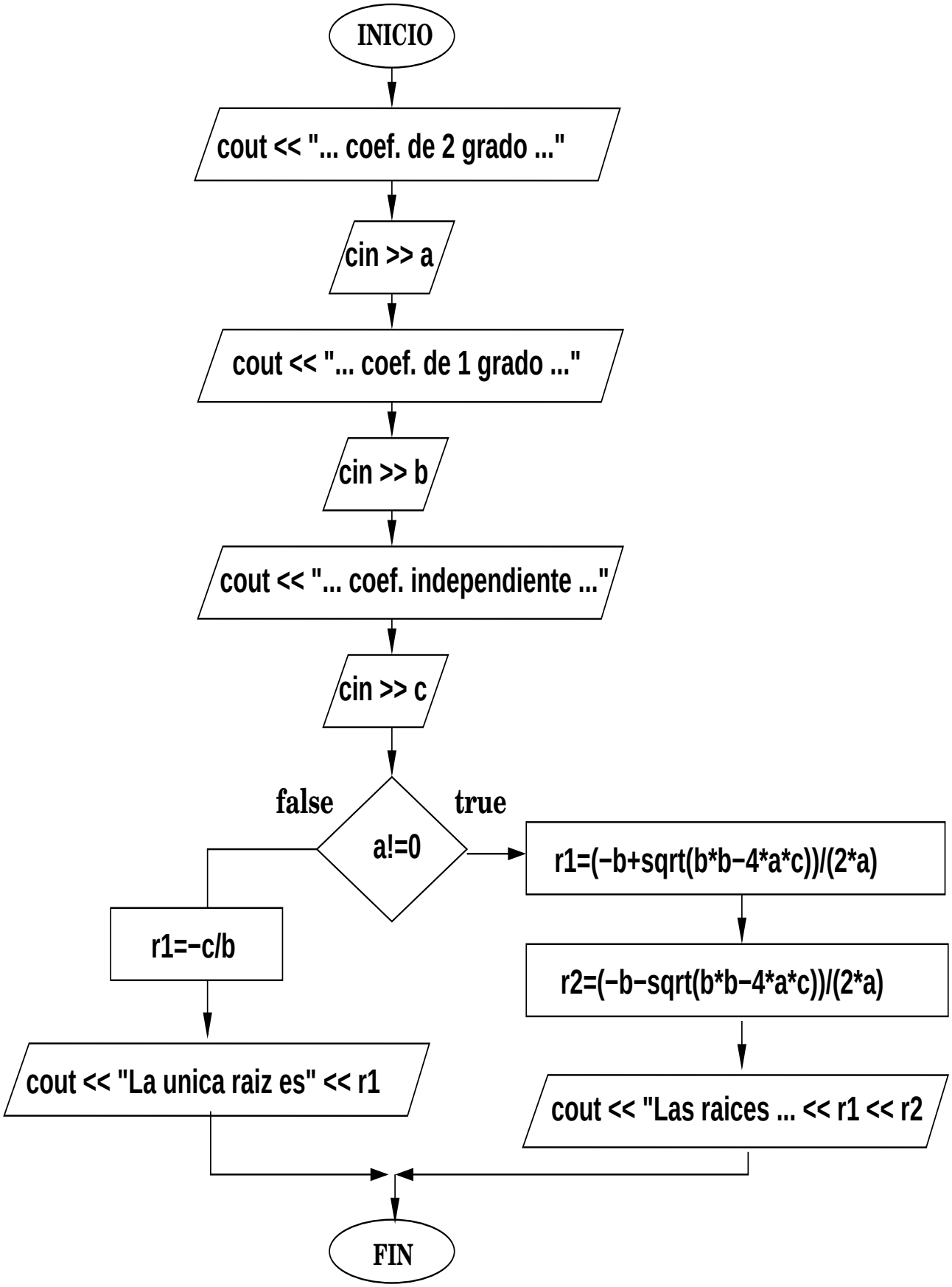
***Problema*: Se evalúan dos condiciones equivalentes.**

***Ejercicio*: Leer dos valores desde teclado e imprimir el mayor.**

2.2.3. CONDICIONAL DOBLE

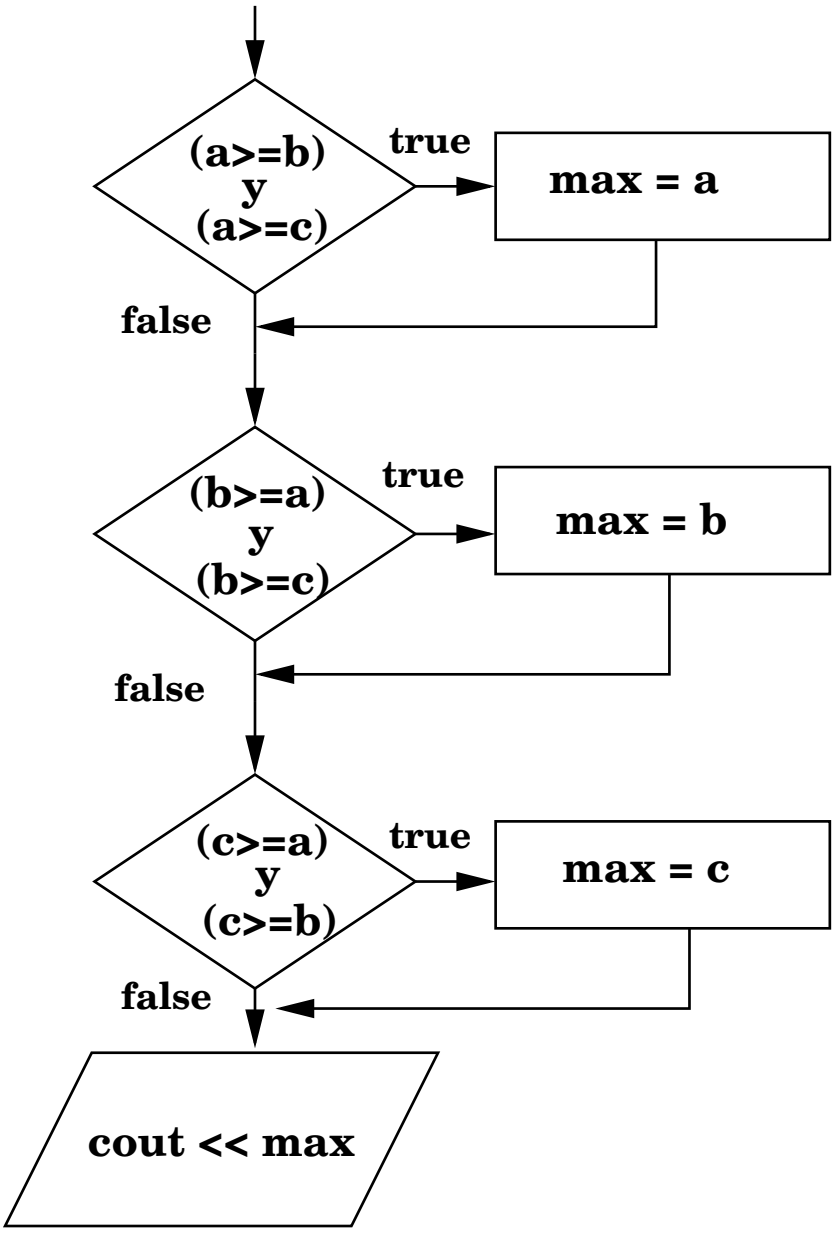
```
if <condición>
  <bloque if>;
else
  <bloque else>;
```




INICIO
cout << "... coef. de 2 grado ..."
cin >> a
cout << "... coef. de 1 grado ..."
cin >> b
cout << "... coef. independiente ..."
cin >> c
false
a!=0
true
r1=(-b+sqrt(b*b-4*a*c))/(2*a)
r2=(-b-sqrt(b*b-4*a*c))/(2*a)
cout << "Las raices ... << r1 << r2
r1=-c/b
cout << "La unica raiz es" << r1
FIN

Ejemplo: El mayor de tres números (primera aproximación)

```
if ((a>=b) && (a>=c))
   max = a;
if ((b>=a) && (b>=c))
   max = b;
if ((c>=a) && (c>=b))
   max = c;
cout << "El mayor es " << max << endl;
```



```
/* Programa que calcula las raíces de una ecuación
   de segundo grado (TERCERA APROXIMACIÓN)
   Entradas: Los parámetros de la ecuación
   Salidas: Las raíces de la parábola
*/ #include <iostream> #include <cmath> using namespace std;

int main(){
  double a,b,c;    // Parámetros de la ecuación
  double r1,r2;    // Raíces obtenidas

  cout << "\nIntroduce coeficiente de 2º grado: ";
  cin >> a;
  cout << "\nIntroduce coeficiente de 1er grado: ";
  cin >> b;
  cout << "\nIntroduce coeficiente independiente: ";
  cin >> c;

  if (a!=0) {
    r1 = ( -b + sqrt( b*b-4*a*c ) ) / (2*a);
    r2 = ( -b - sqrt( b*b-4*a*c ) ) / (2*a);
    cout << "Las raíces son" << r1 << " y "
         << r2 << endl;
  }
  else {
    r1=-c/b;
    cout << "La única raíz es " << r1 << endl;
  }
}
```

2.2.4. ANIDAMIENTO DE ESTRUCTURAS CONDICIONALES

Dentro de un bloque if (else), puede incluirse otra estructura condicional, anidándose tanto como permita el compilador.

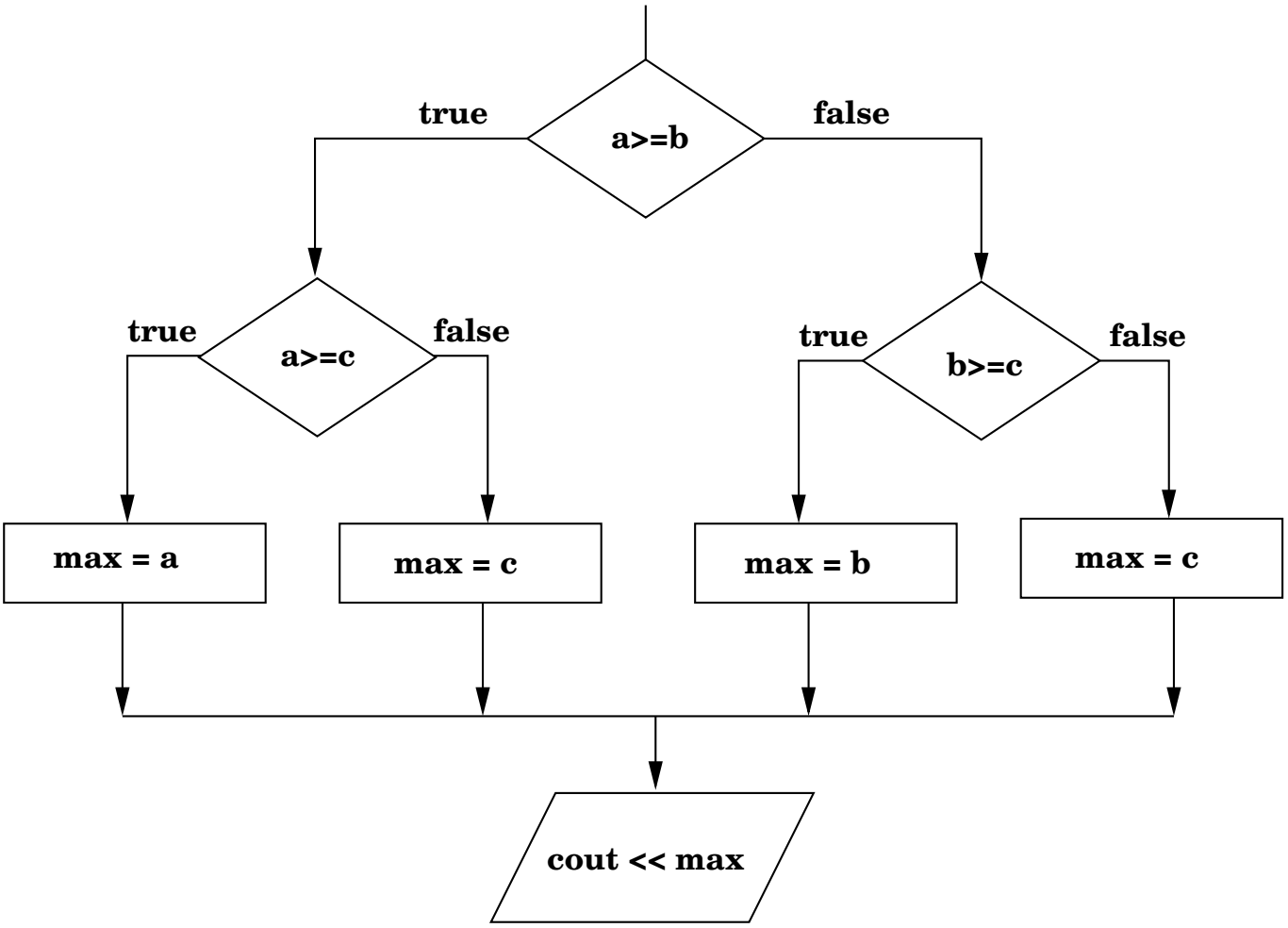
```
if (condic_1) {
   inst_1;

   if (condic_2) {
      inst_2;
   }
   else {
     inst_3;
   }

   inst_4;
}
else {
    inst_5;
}
```

¿Cuándo se ejecuta cada instrucción?

	condic_1	condic_2
inst_1	**true**	**independiente**
inst_2	**true**	**true**
inst_3	**true**	**false**
inst_4	**true**	**independiente**
inst_5	**false**	**independiente**

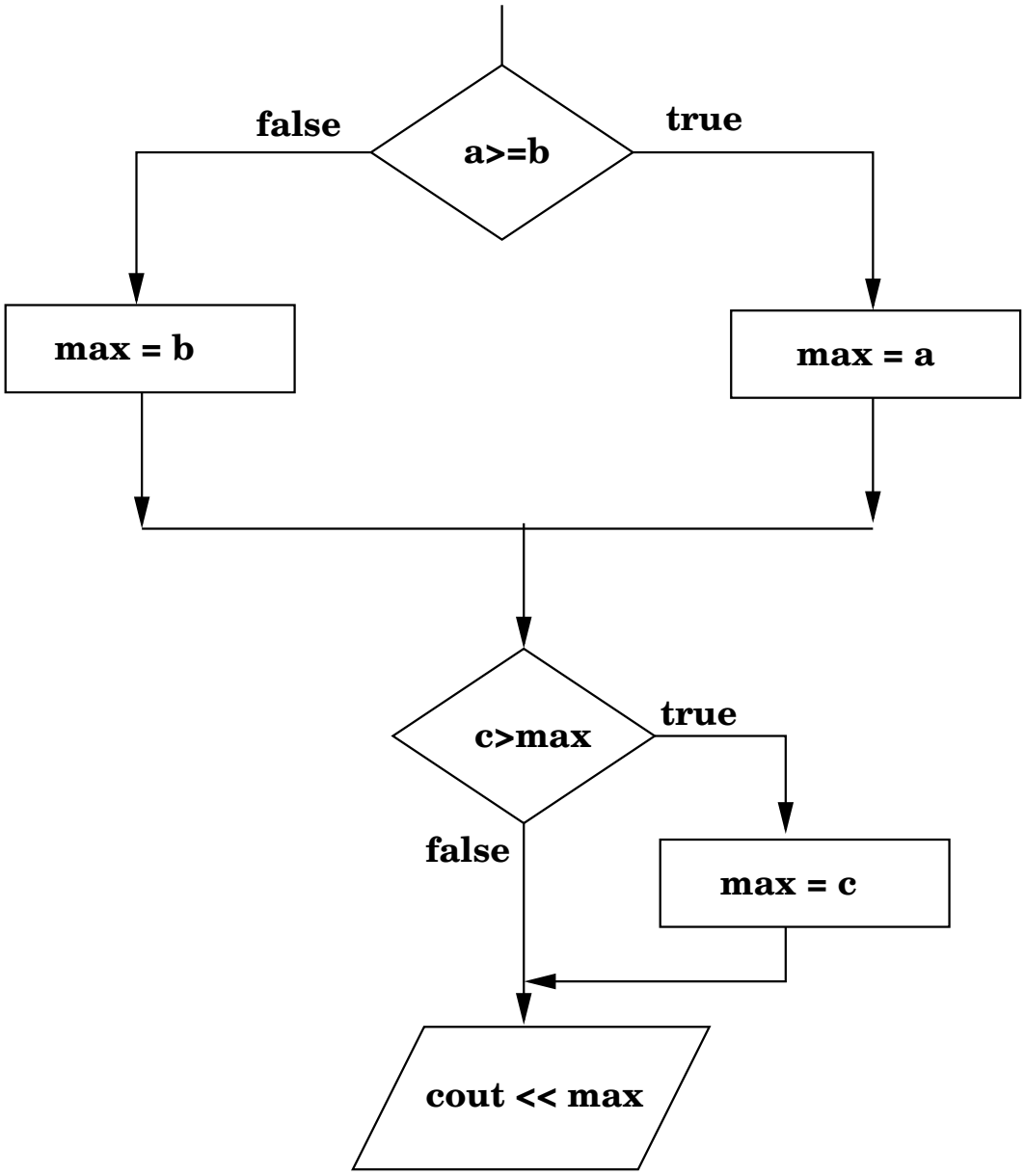
Ejemplo: El mayor de tres números (segunda aproximación)

```
if (a>=b)
   if (a>=c)
      max = a;
   else
      max = c;
else
   if (b>=c)
      max = b;
   else
      max = c;

cout << "El mayor es " << max << endl;
```



Ejemplo: El mayor de tres números (tercera aproximación)

```
if (a>=b)
   max = a;
else
   max = b;
if (c>max)
   max = c;
cout << "El mayor es " << max << endl;
```



¿Con qué solución nos quedamos?
Supongamos que todos los valores son distintos

- **La solución 1 siempre evalúa 3 condiciones y realiza 1 asignación.**
- **La solución 2 siempre evalúa 2 condiciones y realiza 1 asignación.**
- **La solución 3 siempre evalúa 2 condiciones y realiza 1 ó 2 asignaciones.**

La solución 2 es más eficiente.

La solución 3 es más ."elegante".

Siempre habrá que llegar a un compromiso entre eficiencia y *elegancia* (legibilidad, concisión rapidez de desarrollo, reutilización, etc...).

El mayor de cuatro números. Aproximación 1:

```
if (a>=b)
   if (a>=c)
      if (a>=d)
         max = a;
      else
         max = d;
   else
      if (c>=d)
         max = c;
      else
         max = d;
else
   if (b>=c)
     if (b>=d)
         max = b;
      else
         max = d;
   else
      if (c>=d)
         max = c;
      else
         max = d;

cout << "El mayor es " << max << endl;
```

Aproximación 2:

```
if (a>=b)
   max = a;
else
   max = b;
if (c>max)
   max = c;
if (d>max)
   max = d;
cout << "El mayor es " << max << endl;
```

Ejemplo: ¿Son equivalentes las siguientes estructuras?

```
if (c1 && c2)
   bloque_A
else
   bloque_B
```

```
if (c1)
   if (c2)
      bloque_A
   else
      bloque_B
else
   bloque_B
```

Primer caso:

```
bloque_A:   c1 true
            c2 true

bloque_B: (c1&&c2) false:
          c1 true  y c2 false
          c1 false y c2 true
          c1 false y c2 false
```

Segundo caso:

```
bloque_A:   c1 true
            c2 true

bloque_B: c1 true y c2 false
          o bien    c1 false
```

Son equivalentes. Elegimos la primera.

Al aumentar el anidamiento se va perdiendo en legibilidad y concisión.

Evaluación en ciclo corto y en ciclo largo:

***Evaluación en ciclo corto*: El compilador optimiza la evaluación de expresiones lógicas evaluando sus términos de izquierda a derecha hasta que sabe el resultado de la expresión completa (lo que significa que puede dejar términos sin evaluar). La mayoría de los compiladores realizan este tipo de evaluación por defecto.**

***Evaluación en ciclo largo*: El compilador evalúa todos los términos de la expresión lógica para conocer el resultado de la expresión completa.**

Ejemplo:

```
if ( (a>0) && (b<0) && (c<1) ) ...
```

Supongamos `a = -3`.

- ***Ciclo largo*: El compilador evalúa (innecesariamente) todas las expresiones lógicas**
- ***Ciclo corto*: El compilador evalúa sólo la primera expresión lógica.**

***Moraleja*: Poner primero aquellas condiciones que sean más probables de evaluarse como `false`.**

Nota: Lo mismo pasa cuando el operador es `||`

Ejemplo: Estructuras condicionales anidadas

```
/* Programa que permite sumar o restar dos números enteros
   Entradas: Dos números enteros dato1, dato2 y la operación a
   realizar (un carácter: 'S' para sumar y 'R' para restar).
   Salidas: El resultado de la operación
*/

#include <iostream>
using namespace std;

int main(){
  int dato1,dato2;
  char opcion;

  cout << "\nIntroduce el primer operando: ";
  cin >> dato1;
  cout << "\nIntroduce el segundo operando: ";
  cin >> dato2;
  cout << "\nSelecciona (S) sumar, (R) restar: ";
  cin >> opcion;

  if (opcion == 'S')
    cout << "Suma = " << dato1+dato2 << endl;
  if (opcion == 'R')
    cout << "Resta = " << dato1-dato2 << endl;
  if (opcion != 'R' && opcion != 'S')
    cout << "Ninguna operación" << endl;
}
```

Ejemplo: Estructuras condicionales anidadas (continuación)

Lo hacemos más eficientemente:

```
  if (opcion == 'S')
    cout << "Suma = " << dato1+dato2 << endl;
  else
      if (opcion == 'R')
         cout << "Resta = " << dato1-dato2 << endl;
      else
         cout << "Ninguna operación" << endl;
}
```

Una forma también válida de tabular:

```
  if (opcion == 'S')
    cout << "Suma = " << dato1+dato2 << endl;
  else if (opcion == 'R')
    cout << "Resta = " << dato1-dato2 << endl;
  else
    cout << "Ninguna operación" << endl;
```

2.2.5. ESTRUCTURA CONDICIONAL MÚLTIPLE

```
switch (<expresión>) {
  case <constante1>:
    <sentencias1>
    break;
  case <constante2>:
    <sentencias2>
    break;
  .....
  [default:
    <sentencias>]
}
```

- *<expresión>* **es un expresión entera.**
- *<constante1>* **ó** *<constante2>* **es un literal tipo entero o tipo carácter.**
- `switch` **sólo comprueba la igualdad.**
- **No debe haber dos casos (**`case`**) con la misma** *<constante>* **en el mismo** `switch`**. Si esto ocurre, sólo se ejecutan las sentencias correspondientes al caso que aparezca primero.**
- **El identificador especial** `default` **permite incluir un caso por defecto, que se ejecutará si no se cumple ningún otro. Se suele colocar como el último de los casos.**
- **En las estructuras condicionales múltiples también se permite el anidamiento.**

Se usa frecuentemente en la construcción de menús:

```
/* Programa que permite sumar o restar dos números enteros
   Entradas: Dos números enteros dato1, dato2 y la operación a
   realizar (un carácter: 'S' para sumar y 'R' para restar).
   Salidas: El resultado de la operación
*/

#include<iostream>
using namespace std;

int main(){
  int dato1,dato2;
  char opcion;

  cout << "\nIntroduce el primer operando: ";
  cin >> dato1;
  cout << "\nIntroduce el segundo operando: ";
  cin >> dato2;
  cout << "\nSelecciona (S) sumar, (R) restar: ";
  cin >> opcion;

  switch (opcion) {
    case 'S': cout << "Suma = " << dato1+dato2 << endl;
              break;
    case 'R': cout << "Resta = " << dato1-dato2 << endl;
              break;
    default:
              cout << "Ninguna operación" << endl;
  }
}
```

Se pueden realizar las mismas operaciones para un grupo determinado de constantes:

```
switch (opcion) {
  case 's':
  case 'S': cout << "Suma = " << dato1+dato2 << endl;
            break;
  case 'r':
  case 'R': cout << "Resta = " << dato1-dato2 << endl;
            break;
  default:
            cout << "Ninguna operación" << endl;
            break;
            // Ponemos break; aunque no sea necesario
}
```

Ejemplo:

```
// Programa que ilustra el comportamiento de switch.

#include<iostream>
using namespace std;

int main() {
  int  indicador;
  char opcion;
  cout << "Escoja una opción del 1 al 5";
  cin >> opcion;
  indicador=-1;

  switch (opcion) {
    case '1':
    case '2':
    case '3': indicador=0;
              break;
    case '4': indicador=1;
    case '5': indicador=indicador*2;
              break;
    default: cout << "Opción incorrecta" << endl;
  }
  cout << indicador;
}
```

¿Cuántas salidas diferentes ofrece este programa?

2.3. ESTRUCTURAS REPETITIVAS

Las *estructuras repetitivas* son también conocidas como *bucles*, *ciclos* o *lazos*.

Una *estructura repetitiva* permite la ejecución de una secuencia de sentencias:

- **o bien, hasta que se satisface una determinada condición (controladas por condición)**
- **o bien, un número determinado de veces (controladas por contador)**

2.3.1. BUCLES CONTROLADOS POR CONDICIÓN: PRE-TEST Y POST-TEST

2.3.1.1. FORMATO

Pre-test

```
while (<condición>) {
  <cuerpo bucle>
}
```

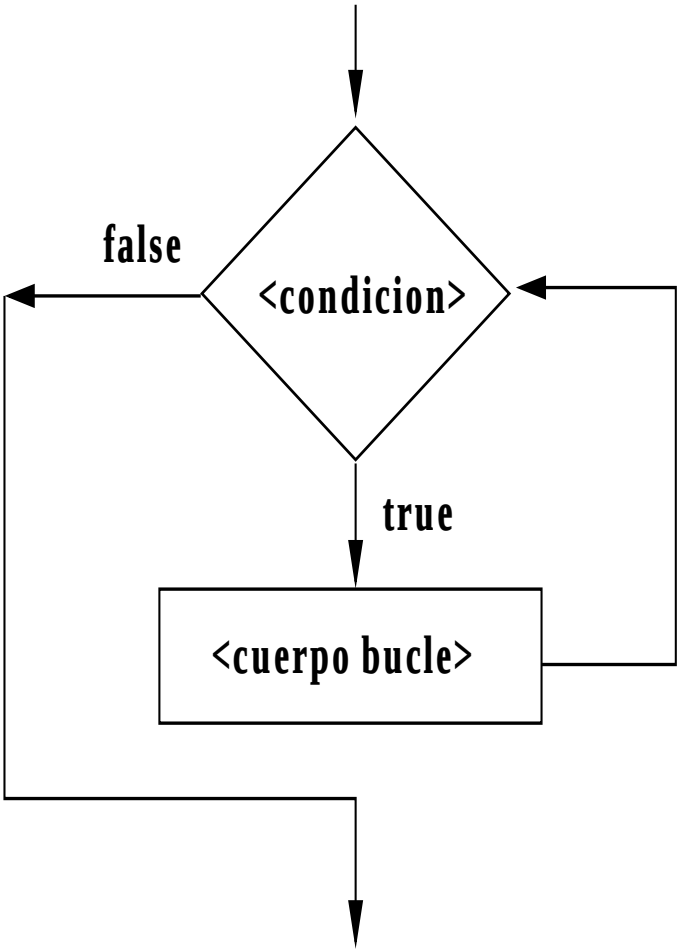
Post-test

```
do {
  <cuerpo bucle>
} while (<condición>);
```

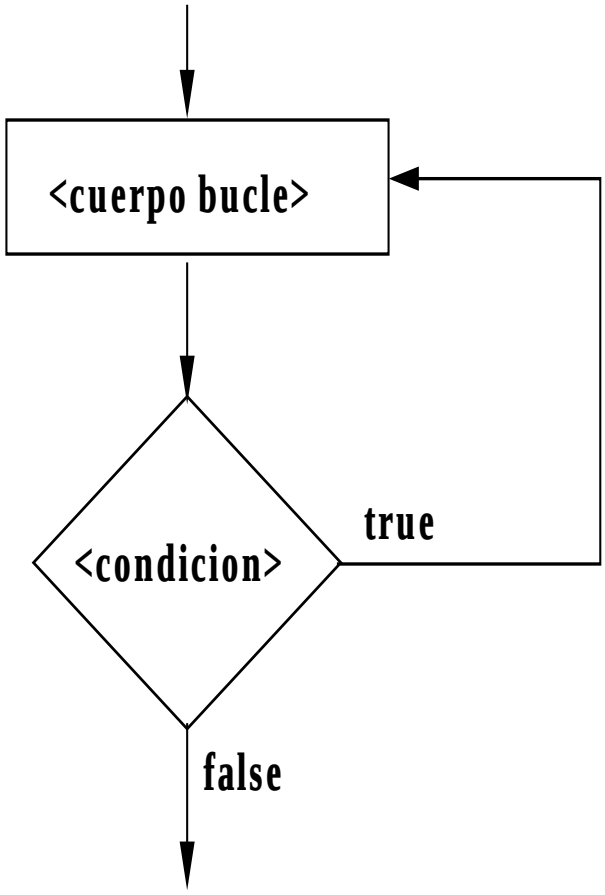
***Funcionamiento*: En ambos, se va ejecutando el cuerpo del bucle mientras la condición sea verdad.**

- **En un bucle `while`, primero se pregunta y luego (en su caso) se ejecuta.**
- **En un bucle `do while`, primero se ejecuta y luego se pregunta.**

(Pre-test)



(Post-test)



Elección Pre-test, o post-test: ¿Quiero que siempre se ejecute al menos una vez el cuerpo?

Ejemplo: Escribir 20 líneas con 5 estrellas cada una.

```
total = 1;

do{
    cout << "*****" << endl;
    total++;
}while (total <= 20);
```

O bien:

```
total = 0;

do{
    cout << "*****" << endl;
    total++;
}while (total < 20);
```

O bien:

```
total = 1;

while (total <= 20){
    cout << "*****" << endl;
    total++;
}
```

O bien:

```
total = 0;

while (total < 20){
    cout << "*****" << endl;
    total++;
}
```

***Ejemplo*: Leer un número positivo `tope` desde teclado e imprimir `tope` líneas con 5 estrellas cada una.**

```
cin >> tope;
total = 0;

do{
    cout << "*****" << endl;
    total++;
}while (total < tope);
```

***Problema*: ¿Qué ocurre si `tope = 0`?**
***Solución*:**

```
cin >> tope;
total = 0;

while (total < tope){
    cout << "*****" << endl;
    total++;
}
```

Ejemplo: Leer un entero `tope` y escribir los pares $\leq$ `tope`

```
cin >> tope;
par = 0;

while (par<= tope){
    par = par+2;
    cout << par;
}
```

¡Cuidado Casos Extremos!
Al final, escribe uno más. Cambiar Orden

```
cin >> tope;
par = 0;                // Primer candidato

while (par <= tope){    // ¿Es bueno?
    cout << par;        // Si => imprimelo
    par = par+2;        //       calcular nuevo
                        //       candidato
}                       // No => Salir
```

Si no queremos que salga 0, cambiaríamos la inicialización:

```
par = 2
```

Ejemplo:

```
// Programa que suma los valores leídos desde teclado
// hasta que se introduzca el valor -1.
#include<iostream>
using namespace std;

int main(){
  int suma, numero;

  suma = 0;

  do{
    cin >> numero;
    suma = suma + numero;
  }while (numero != -1);

  cout << "La suma es " << suma << endl;
}
```

Problema: Procesa el '-1'y lo suma.

Solución:

```
  do{
    cin >> numero;
    if (numero != -1)
      suma = suma + numero;
  }while (numero != -1);
```

Funciona, pero evalúa dos veces la misma condición.

Solución: técnica de lectura anticipada.

2.3.1.2. LECTURA ANTICIPADA

Antes de entrar a un bucle pre-test, se comprueba el primer candidato.

***Ejemplo*:**

```
// Programa que suma los valores leídos desde teclado
// hasta que se introduzca el valor -1.
// Usamos ciclo pre-test y lectura anticipada:

#include <iostream>
using namespace std;

int main(){
  int suma, numero;

  suma = 0;

  cin >> numero;              // Primer candidato
  while (numero != -1) {      // ¿Es bueno?
    suma = suma + numero;
    cin >> numero;            // Leer siguiente candidato
  }

  cout << "La suma es " << suma << endl;
}
```

Ejemplo:

```
// Programa que va leyendo números enteros hasta que se
// introduzca el cero. Imprimir el número de pares e
// impares introducidos.

#include <iostream>
using namespace std;

int main(){
   int ContPar, ContImpar, valor;

   ContPar=0;
   ContImpar=0;
   cout << "Introduce valor: " << endl;
   cin >> valor;

   while (valor != 0) {
      if (valor % 2 == 0)
         ContPar = ContPar + 1;
      else
         ContImpar = ContImpar + 1;

      cout << "Introduce valor: " << endl;
      cin >> valor;
   }

   cout << "Fueron " << ContPar << " pares y "
        << ContImpar << " impares" << endl;
}
```

2.3.1.3. BUCLES SIN FÍN

Ejercicio:¿Cuantas iteraciones se producen?

```
contador = 2;

while (contador < 3) {
    contador--;
    cout << contador << endl;
}
```

```
contador = 1;

while (contador != 10) {
    contador = contador + 2;
}
```

Moraleja: Fomentar el uso de condiciones de desigualdad.

```
contador = 1;

while (contador <= 10) {
    contador = contador + 2;
}
```

2.3.1.4. CREACIÓN DE FILTROS CON CICLOS POST-TEST

Objetivo: Leer un valor y no permitir al usuario que lo introduzca fuera de un rango determinado.

Una de las pocas veces que se usarán ciclos post-test.

Ejemplo:

```
// Introducir un único valor, pero positivo.
// Imprimir el coseno.
// Entrada: Un valor entero positivo.
// Salida: El coseno del valor introducido.

#include <iostream>
#include <cmath>
using namespace std;

 int main(){
 int valor;

 do {
     cout << "\nIntroduzca un valor positivo: ";
     cin >> valor;
 }while (valor < 0);

 cout << cos(valor);
}
```

Ejemplo: Leer una opción de un menú. Sólo se admite s ó n.

```
char opcion;
do{
   cout << "Introduzca una opción: " << endl;
   cin >> opcion;
}while ((opcion!='s') || (opcion!='S') ||
        (opcion!='n') || (opcion!='N') );
```

```
char opcion;
do{
   cout << "Introduzca una opción: " << endl;
   cin >> opcion;
}while ((opcion!='s') && (opcion!='S') &&
        (opcion!='n') && (opcion!='N') );
```

Ejercicio: Leer dos valores desde el teclado forzando a que ambos sean distintos de cero.

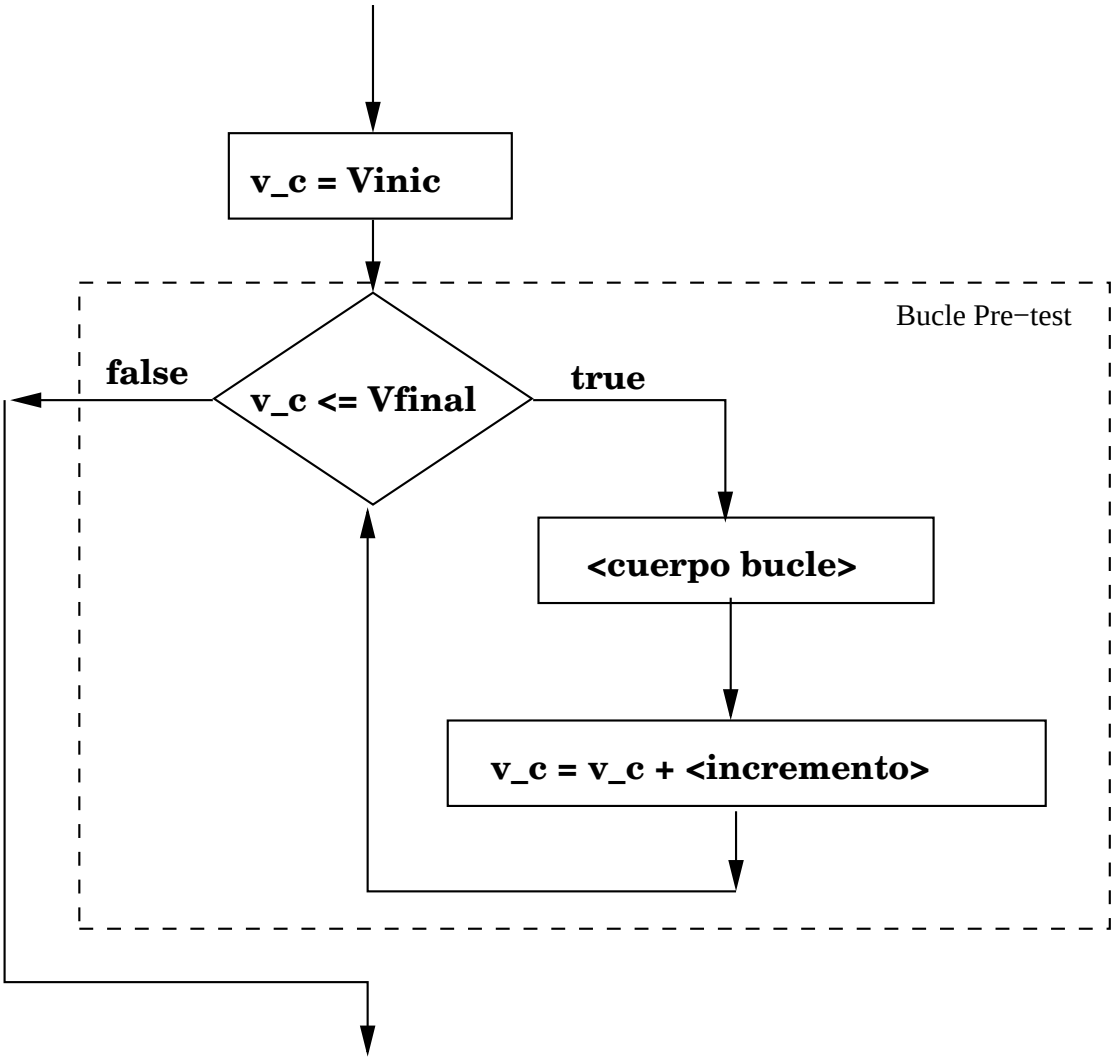
2.3.1.5. BUCLES CONTROLADOS POR CONTADOR

Se utilizan para repetir un conjunto de sentencias un número fijo de veces. Se necesita una variable controladora, un valor inicial, un valor final y un incremento.

***Ejemplo*: Hallar la media aritmética de cinco enteros.**

```
// Hallar la media de 5 números enteros.
// Entradas: 5 números enteros (se leen en valor).
// Salidas: La media de los 5 números.
#include <iostream>
using namespace std;

int main(){
   int i, valor, suma;
   double media;

   suma = 0;
   i = 1;                  // v_c =  Vinic

   while (i <= 5){         // v_c <= Vfinal
     cout << "Introduce el número " << i << "º: \n";
     cin >> valor;
     suma = suma + valor;

     i++;                  // v_c = v_c + 1
   }
   media = suma / 5.0;
   cout << "La media es " << media << endl;
}
```

Diagrama de flujo



`v_c` es la variable controladora del ciclo.

`Vinic` valor inicial que toma `v_c`.

`Vfinal` valor final que debe tomar `v_c`.

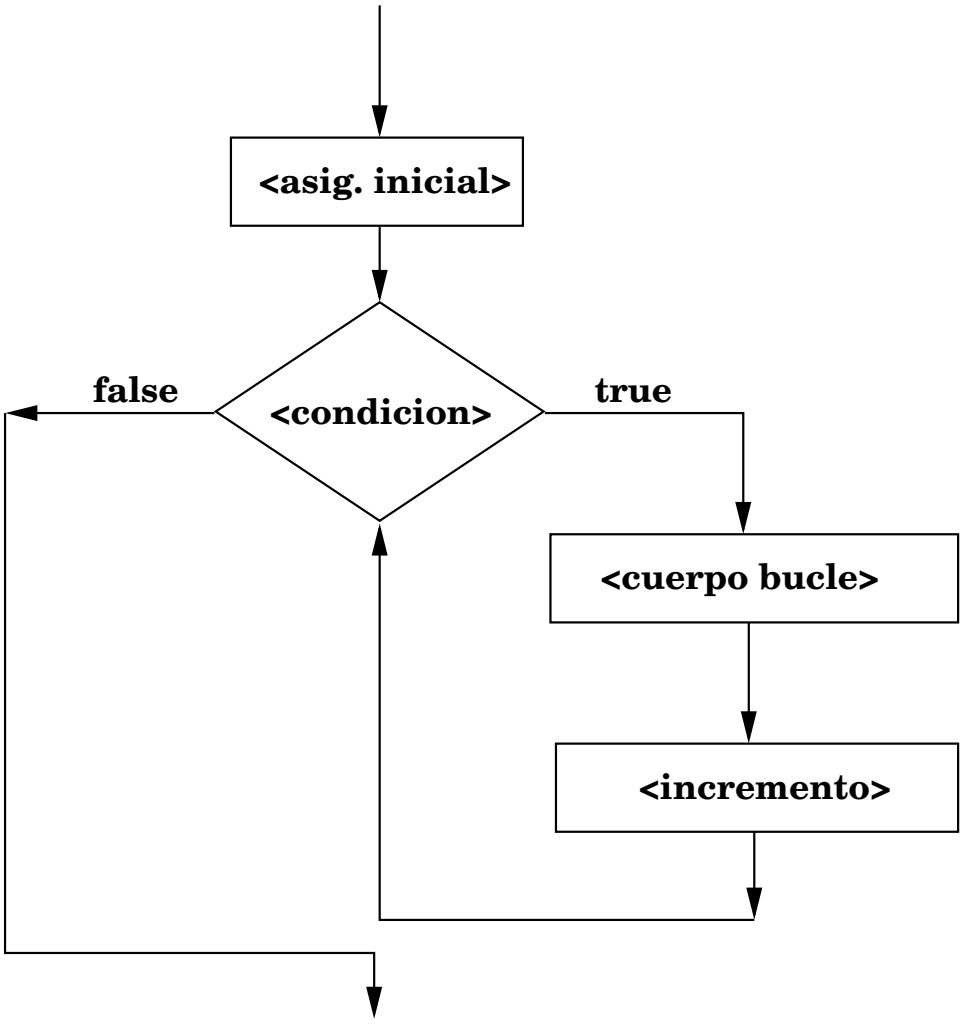
2.3.2. BUCLE `for` COMO BUCLE CONTROLADO POR CONTADOR

2.3.2.1. FORMATO

La sentencia `for` permite la construcción de una forma compacta de los ciclos controlados por contador, aumentando la legibilidad del código.

```
// Hallar la media de 5 números enteros.
// Ejemplo de uso del for.

#include <iostream>
using namespace std;

int main(){
   int i, valor, suma;
   double media;

   suma = 0;

   for (i=1 ; i <= 5 ; i++){
     cout << "Introduce el número " << i << "º: \n";
     cin >> valor;
     suma = suma + valor;
   }

   media = suma / 5.0;
   cout << "La media es " << media << endl;
}
```

`for` **([<*asig.inicial*>];[<*condicion*>];[<*incremento*>])**
<*cuerpo bucle*>



- **<*asig. inicial*> asignación del valor inicial a la variable controladora del ciclo.**

  `v_c = Vinic`

- **<*incremento*> determina el modo en que la variable controladora cambia su valor para la siguiente iteración.**

  - **Incrementos positivos:** `v_c = v_c + incremento`
  - **Incrementos negativos:** `v_c = v_c - incremento`

- **<*condicion*> determina cuándo ha de continuar la ejecución del cuerpo del bucle.**

  - `v_c <= Vfinal` **ó bien** `v_c < Vfinal` **para incrementos positivos**
  - `v_c >= Vfinal` **ó bien** `v_c > Vfinal` **para incrementos negativos**

```
for (v_c = Vinic ; v_c <= Vfinal ; v_c++)
    <cuerpo bucle>
```

***Ejemplo*: Imprimir 9 veces el mensaje** `Hola`

```
int i;

for (i = 1; i <= 9; i++)
   cout << "Hola \n";
```

¿Con qué valor sale la variable `i`**? 10**

Cuando termina un bucle for, la variable contadora se queda con el primer valor que hace que la condición del bucle sea falsa.

¿Cuántas iteraciones se producen en un `for`?

- **Si** `incremento = 1`, `Vinic` **es menor que** `Vfinal` **y la condición es** `v_c <= Vfinal`

  `Vfinal - Vinic + 1`

- **Si** `incremento = 1`, `Vinic` **es menor que** `Vfinal` **y la condición es** `v_c < Vfinal`

  `Vfinal - Vinic`

```
for (i = 0; i < 9; i++)
   cout << "Hola \n";
```

```
for (i = 9; i > 0; i--)
    cout << "Hola \n";
```

Número de iteraciones con incrementos cualesquiera.

Si es del tipo `variable < valor_final`**, tenemos que contar cuantos intervalos de longitud igual a** `incremento` **hay entre los valores inicial y final. Como es un menor estricto, debemos quedarnos justo en el elemento anterior al final, es decir,** `valor_final-1`**. El número de intervalos será**

```
(valor_final-1-valor_inicial)/2
```

Y el número de iteraciones será

```
(valor_final-1-valor_inicial)/2 + 1
```

Si fuese `variable <= valor_final`**, el número de iteraciones sería:** `(valor_final-valor_inicial)/2 + 1`

2.3.2.2. ALGUNAS APLICACIONES

Ejemplo: ¿Cuántos pares hay en $[-10, 10]$?

```
int i, contador;

contador = 0;

for(i = -10; i <= 10; i++) {
  if (i % 2 == 0)
      contador++;
}
cout << contador << end;
```

Es un ejemplo de _bucle contador_, pues cuenta el número de veces que ocurre una condición durante el bucle.

Ejemplo: Sumar los valores menores o iguales que `Tope`, es decir, $\sum_{i=1}^{\text{Tope}} i$

```
// Suma los términos de la sucesión 1+2+3+...+Tope
// Entrada: El valor de Tope
// Salida: La suma de los elementos de la sucesión

#include <iostream>
using namespace std;

int main(){
  int i;
  int suma;  /* acumulador */
  int Tope;  /* limite superior */

  cout << "Introduce el limite superior \n";
  cin >> Tope;

  suma = 0;
  for (i=1; i<=Tope; i++)
    suma = suma + i;

  cout << "El resultado es "<< suma << endl;
}
```

Es un ejemplo de *bucle acumulador*, pues va acumulando en una variable el resultado de una expresión que se calcula en cada iteración.

2.3.2.3. CRITERIO DE USO DE BUCLE `for`

Únicamente mirando la cabecera de un bucle `for` sabemos cuantas iteraciones se van a producir. Por eso, en los casos en los que sepamos de antemano cuantas iteraciones necesitamos, usaremos un bucle `for`. En otro caso, usaremos un bucle `while` ó `do while`.

Para mantener dicha finalidad, es necesario respetar la siguiente restricción:

No se debe modificar el valor de la variable controladora, ni el valor final dentro del cuerpo del bucle.

***Ejemplo*. Sumar pares hasta un tope**

```
suma  = 0;

for (par=2; par<=tope ; par++){
   if (par%2 == 0)
      suma = suma+valor;
   else
      par++;   // :-(
}
```

```
N = 5;

for (i = 1; i <= N; i++) {
  cout << i << endl;       //      NN  N  OOOOO
  if (N % 2 !=0)           //      N N N  O   O
    N++;                   // <--  N  NN  OOOOO
}
```

```
N = 100;

for (i = 0; i < N; i++) {
  cout << i << endl;
                           //      NN  N  OOOOO
  if (i % 13 !=0)          //      N N N  O   O
    i = N;                 // <--  N  NN  OOOOO
}
```

2.3.3. EL BUCLE `for` COMO CICLO CONTROLADO POR CONDICIÓN

Si bien en muchos lenguajes tales como PASCAL, FORTRAN o BASIC el comportamiento del ciclo `for` es semejante al descrito en la sección anterior, existen otros lenguajes (como es el caso de C y C++) donde este tipo de ciclo no es un ciclo controlado por contador, sino que es un ciclo controlado por condición.

En el caso concreto de C++, su sintaxis es la siguiente:

```
for ([<sentencia inicial>];
     [<expresion booleana>];
     [<sentencia actualizacion>])
    <cuerpo bucle>
```

donde,

- ***<sentencia inicial>*** **es la sentencia de inicialización de la variable controladora,**
- ***<expresion booleana>*** **es cualquier expresión lógica que verifica si el ciclo debe terminar o no,**
- ***<sentencia actualizacion>*** **actualiza el valor de la variable controladora después de cada ejecución de** ***<cuerpo bucle>*****.**

Por tanto, la condición impuesta en el ciclo no tiene por que ser de la forma `v_c` $<$ `Vfinal`, sino que puede ser cualquier tipo de condición.

Esto implica que se puede construir con una sentencia `for` un ciclo donde no se conoce a priori el número de iteraciones que debe realizar el cuerpo del ciclo.

***Ejemplo*: Construir un programa que indique el número de valores que introduce un usuario hasta que se encuentre con un cero.**

```
// Programa para contar el numero de valores que se
// introducen hasta que se encuentra un cero.
//  --- Usando un ciclo while ---

#include <iostream>
using namespace std;

int main(){
  int i, valor;

  cin >> valor;
  i=0;                    // inicializacion

  while (valor !=0){      // condicion
    cin >> valor;
    i++;                  // incremento
  }

  cout << "El número de valores introducidos es " << i;
}
```

```
// Programa para contar el número de valores que se
// introducen hasta que se encuentra un cero.
//  --- Usando un ciclo for ---

#include <iostream>
using namespace std;

int main(){
  int i, valor;

  cin >> valor;

  for (i=0; valor!=0; i++)
    cin >> valor;

  cout << "El número de valores introducidos es " << i << endl;
}
```

REFLEXIÓN: ¿Cuando usar un ciclo `for` como ciclo controlado por condición?

- **Cuando el flujo de control del ciclo sea semejante al de un ciclo controlado por contador,**
- **se preserve la norma de buena práctica de programación de los ciclos `for`, es decir, sin modificar en el cuerpo del ciclo la variable controladora y**
- **resulte más legible que hacerlo con otro tipo de ciclo.**

Ejemplo. Comprobar si un número es primo.

```
#include <iostream>
#include <cmath>
using namespace std;

int main(){
  int valor, divisor;
  bool es_primo;

  cout << "Introduzca un numero natural: ";
  cin >> valor;

  es_primo = true;

  for (divisor = 2 ;
       divisor < valor ;
       divisor++)
    if (valor % divisor == 0)
      es_primo = false;

  if (es_primo)
    cout << valor << " es primo\n";
  else
    cout << valor << " no es primo\n";
}
```

Para hacerlo más eficiente, nos salimos en cuanto sepamos que no es primo.

```
es_primo = true;

for (divisor = 2 ;
     divisor < valor ;
     divisor++)
  if (valor % divisor == 0){
    es_primo = false;
    divisor = valor;          // :-((
  }
```

Nos salimos del bucle con un `bool`. La misma variable `es_primo` nos sirve:

```
for (divisor = 2 ;
     divisor < valor && es_primo ;
     divisor++)
  if (valor % divisor == 0)
    es_primo = false;
```

Incluso podríamos quedarnos en `sqrt(valor)`:

```
// Programa para comprobar si un número es primo

#include <iostream>
#include <cmath>
using namespace std;

int main(){
  int valor, divisor;
  bool es_primo;
  double tope;

  cout << "Introduzca un numero natural: ";
  cin >> valor;

  es_primo = true;
  tope = sqrt(valor);

  for (divisor = 2 ;
       divisor <= tope && es_primo ;
       divisor++)
    if (valor % divisor == 0)
      es_primo = false;

  if (es_primo)
    cout << valor << " es primo\n";
  else
    cout << valor << " no es primo\n";
}
```

Nota: Realmente, para que compile sin problemas debemos usar:

```
tope = sqrt(1.0*valor);
```

***Ejercicios*: ¿Qué salida producen los siguientes trozos de código?**

```
int i, j, total;

total = 0;

for(i=1 ; i<=10; i++)
    total = total + 3;
cout << total << endl;
```

```
total = 0;

for (i=4 ; i<=36 ; i=i+4)
   total++;
```

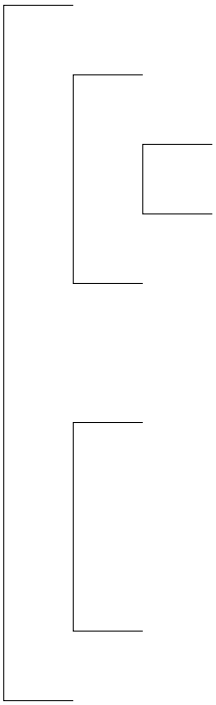
```
total = 0;
i = 4;

while (i <= 36){
    total++;
    i = i+4;
}
```

2.3.4. ANIDAMIENTO DE BUCLES

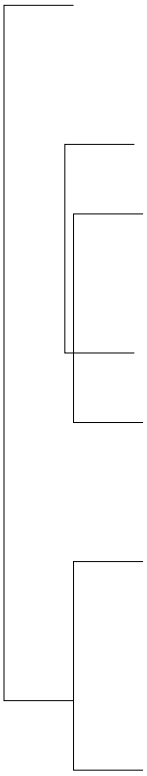
Dos bucles se encuentran anidados, cuando uno de ellos está en el bloque de sentencias del otro.

En principio no existe límite de anidamiento, y la única restricción que se debe satisfacer es que deben estar completamente inscritos unos dentro de otros.

ANIDAMIENTO
PERMITIDO



ANIDAMIENTO
NO PERMITIDO



Ejemplo:

```
// Programa que imprime la tabla de multiplicar de
// los N primeros números.

#include <iostream>
using namespace std;

int main(){
  const int N=3;
  int i,j;

  for (i=1 ; i<=N ; i++) {
    for (j=1 ;j<=N ; j++)
      cout << i << "*" << j <<"=" << i*j << "  ";
    cout << endl;
  }
}
```

Salida

	j=1	**j=2**	**j=3**
i=1	1*1 = 1	1*2 = 2	1*3 = 3
i=2	2*1 = 2	2*2 = 4	2*3 = 6
i=3	3*1 = 3	3*2 = 6	3*3 = 9

¿Qué salida producen los siguientes trozos de código?

```
iteraciones=0;

for (i=0; i<3; i++)
  for (j=0; j<2 ; j++)
    iteraciones++;
```

```
iteraciones=0;

for (i=0; i<4; i++)
  for (j=i; j<2 ; j++)
    iteraciones++;
```

```
 total = 3;

 for (i=2 ; i<=10 ; i=i+2)
     for (j=1 ; j<=8 ; j++)
         total++;
 cout << total << endl;
```

Muy Importante

Existen otras sentencias en la mayoría de los lenguajes que permiten alterar el flujo normal de un programa.

En concreto, en C++ existen las siguientes sentencias:

- `goto`
- `continue`
- `break`

Durante los 60, quedo claro que el uso incontrolado de sentencias de transferencia de control era la principal fuente de problemas para los grupos de desarrollo de software.

Fundamentalmente, el responsable de este problema era la sentencia `goto` que le permite al programador transferir el flujo de control a cualquier punto del programa.

Esta sentencia aumenta considerablemente la complejidad tanto en la legibilidad como en la depuración del código y atenta contra los paradigmas de la PROGRAMACIÓN ESTRUCTURADA.

Por consiguiente, no se usará ninguna de las sentencias anteriores, excepto la sentencia `break` con el propósito aquí descrito dentro de la sentencia `switch`.