Índice general

TEMA 2. ESTRUCTURAS DE CONTROL

Estructuras { **Secuencial**, **Condicional**, **Repetitiva**

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

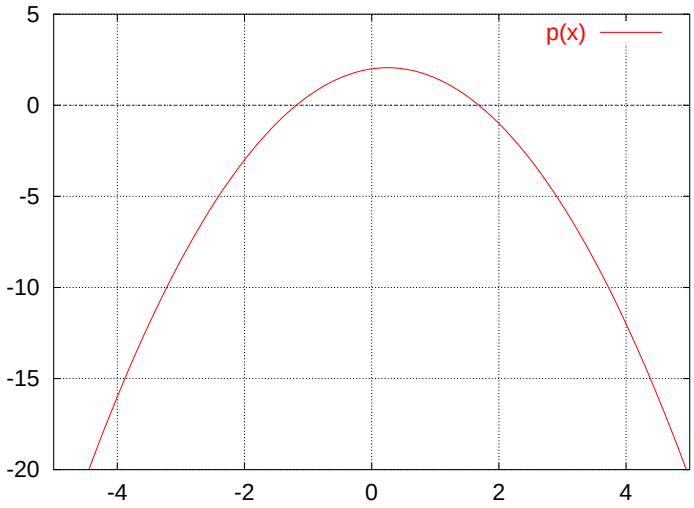
***Ejemplo.* Calcular las raíces de una ecuación de 2º grado.**

***Algoritmo*:**

- ***Entradas*: Los parámetros de la ecuación a, b, c.**
  ***Salidas*: Las raíces de la parábola $r1, r2$**
- ***Descripción*:**
  Calcular $r1, r2$ en la forma siguiente:

$$r_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \qquad r_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

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



```
/* Programa que calcula las raíces de una ecuación
   de segundo grado  (PRIMERA APROXIMACIÓN)
*/
#include <stdio.h>
#include <math.h>


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

1  printf("\nIntroduce coeficiente de 2º grado: ");
2  scanf("%f", &a);
3  printf("\nIntroduce coeficiente de 1er grado: ");
4  scanf("%f", &b);
5  printf("\nIntroduce coeficiente independiente: ");
6  scanf("%f", &c);
7  r1 = ( -b + sqrt( b*b-4*a*c ) ) / (2*a);
8  r2 = ( -b - sqrt( b*b-4*a*c ) ) / (2*a);
9  printf("Las raíces son %f y %f\n", r1, r2);
}
```

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.**

```
 /* Programa que calcula las raíces de una ecuación
    de segundo grado  (SEGUNDA APROXIMACIÓN)
 */
 #include <stdio.h>
 #include <math.h>

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

1  printf("\nIntroduce coeficiente de 2º grado: ");
2  scanf("%f", &a);
3  printf("\nIntroduce coeficiente de 1er grado: ");
4  scanf("%f", &b);
5  printf("\nIntroduce coeficiente independiente: ");
6  scanf("%f", &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     printf("Las raíces son %f y %f\n", r1, r2);
   }
 }
```

2.1. Estructura Condicional

2.1.1. Introducción

Una *condición* es una expresión lógica.

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*.

2.1.2. Estructura Condicional Simple

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

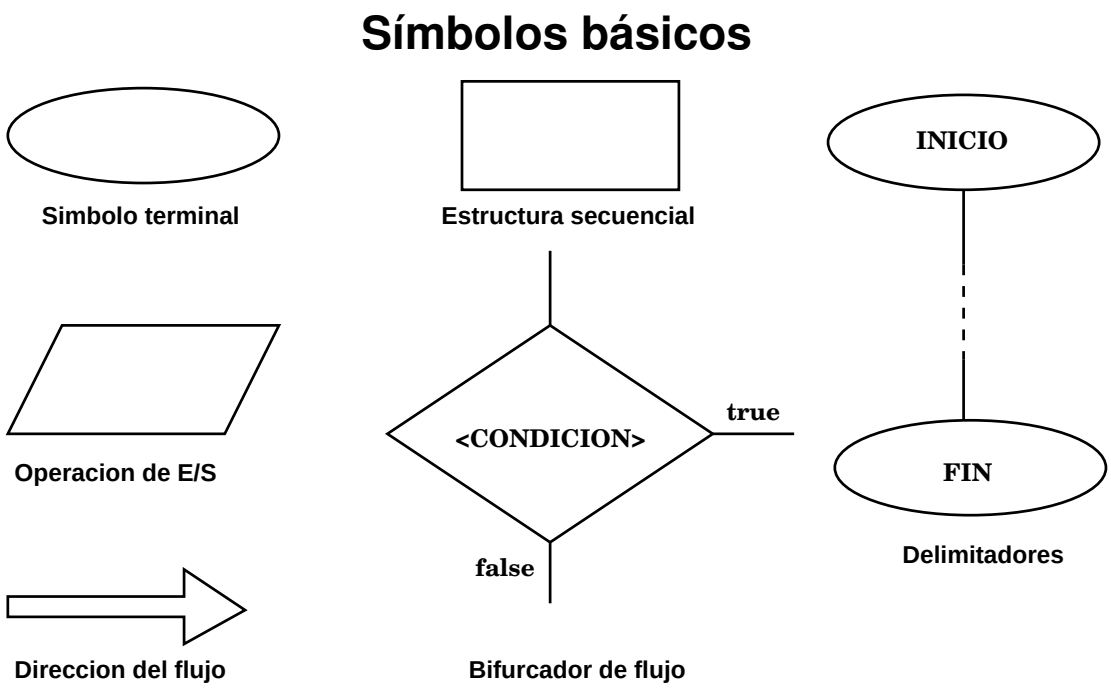
<*condición*> es una expresión lógica

<*bloque if*> es el bloque que se ejecutará si la expresión lógica se evalúa a `true`.

***Nota*. Los paréntesis sobre la condición son obligatorios.**

2.1.3. UNA NOTACIÓN PARA DESCRIBIR ALGORITMOS

Diagramas de flujo:



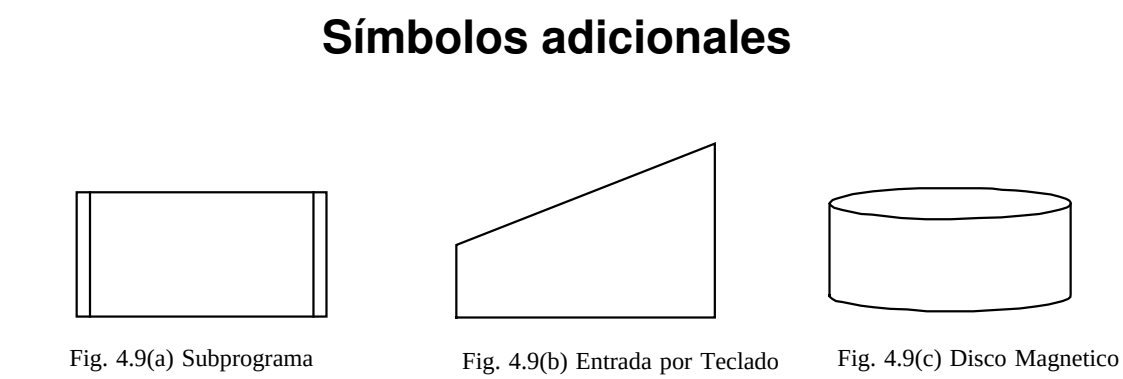


Fig. 4.9(a) Subprograma

Fig. 4.9(b) Entrada por Teclado

Fig. 4.9(c) Disco Magnetico

Fig. 4.9(d) Impresora

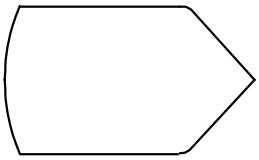


Fig. 4.9(e) Pantalla

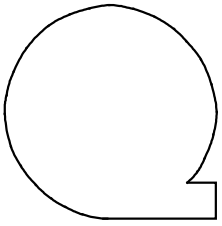
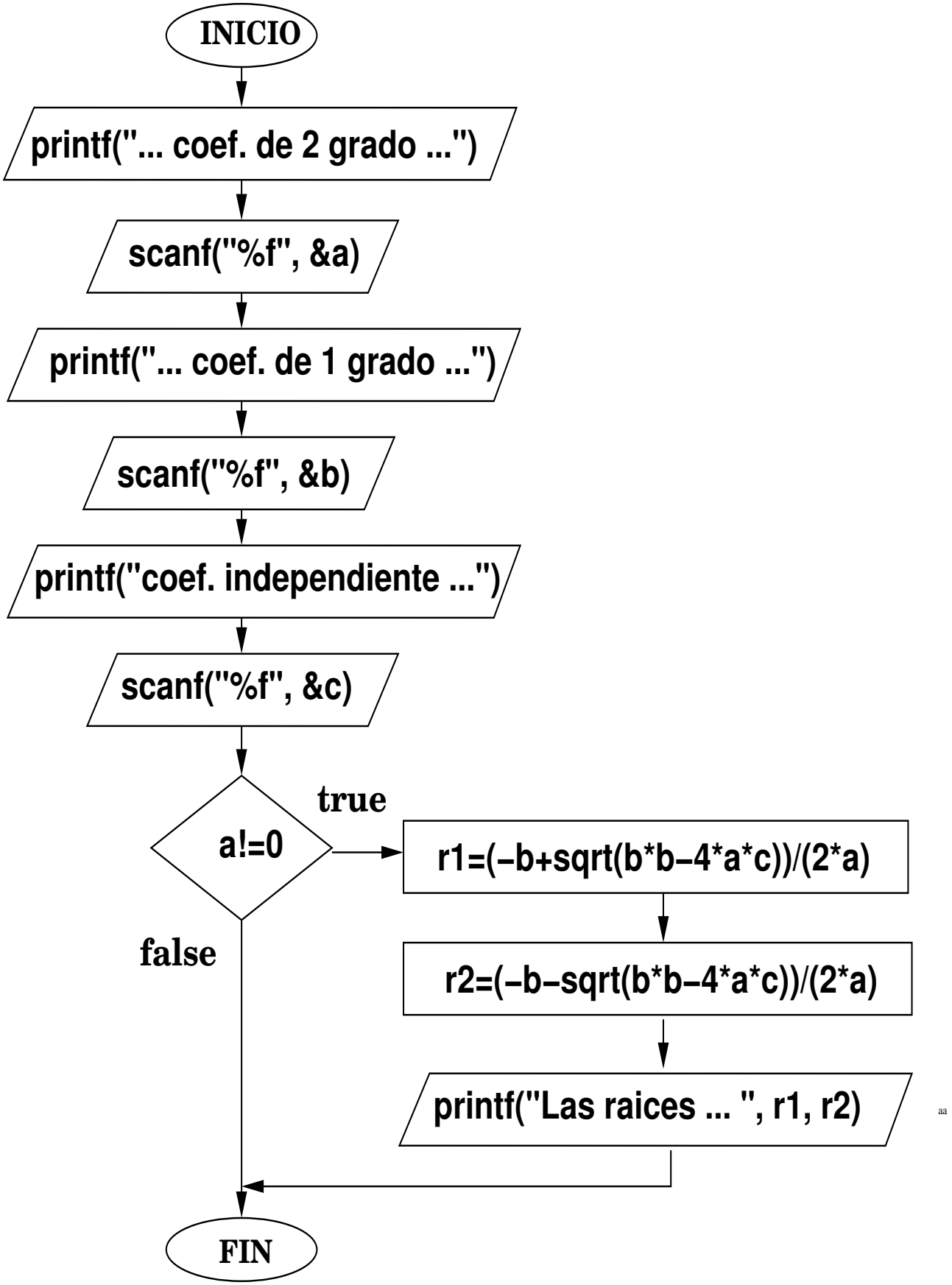


Fig. 4.9(f) Cinta Magnetica

Calcular las raíces de una ecuación de segundo grado



aa

Ejemplo.

```
int edad;
scanf("%d", &edad);

if (edad>18)
   printf("\nEs mayor de edad");
```

o si se prefiere:

```
int edad;
int es_mayor_edad;

scanf("%d", &edad);
es_mayor_edad = (edad>18);

if (es_mayor_edad == 1)
   printf("\nEs mayor de edad");
```

Lo anterior equivale a:

```
if (es_mayor_edad)
   printf("\nEs mayor de edad");
```

Al igual que

```
if (es_mayor_edad==0)
   printf("\nEs menor de edad");
```

equivale a

```
if (!es_mayor_edad)
   printf("\nEs menor de edad");
```

Ejemplo. **Comprobad si es menor de edad o es mayor de 65, pero en cualquier caso con unos ingresos menores a 40000**

```
int edad = 20, ingresos = 23000;
if (edad<18 || edad>=65 && ingresos<40000)    /* :-( */
   printf("\nContribuyente de especial tratamiento");
```

Precedencia de operadores:

```
if ( (edad<18 || edad>=65) && ingresos<40000)  /* :-) */
   printf("\nContribuyente de especial tratamiento");
```

Si hay varias sentencias, es necesario encerrarlas entre llaves. Dicho de otra forma: si no ponemos llaves, el compilador entiende que la única sentencia del bloque `if` es la que hay justo debajo.

```
if (a!=0)
   r1 = ( -b + sqrt( b*b-4*a*c ) ) / (2*a);
   r2 = ( -b - sqrt( b*b-4*a*c ) ) / (2*a); /* Error lógico */
   printf("Las raíces son %f y %f\n", r1, r2);
```

Para el compilador es como si fuese:

```
if (a!=0){
   r1 = ( -b + sqrt( b*b-4*a*c ) ) / (2*a);
}

r2 = ( -b - sqrt( b*b-4*a*c ) ) / (2*a); /* Error lógico */
printf("Las raíces son %f y %f\n", r1, r2);
```

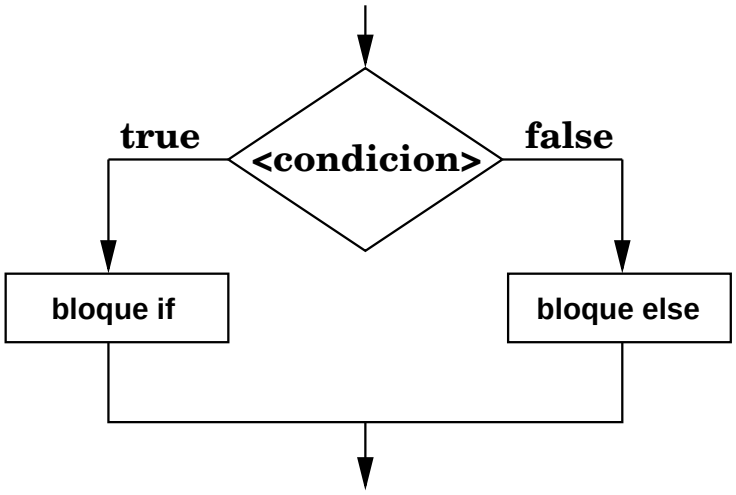
¿Qué pasa si $a = 0$ en la ecuación de segundo grado?
El algoritmo debe devolver $-c/b$

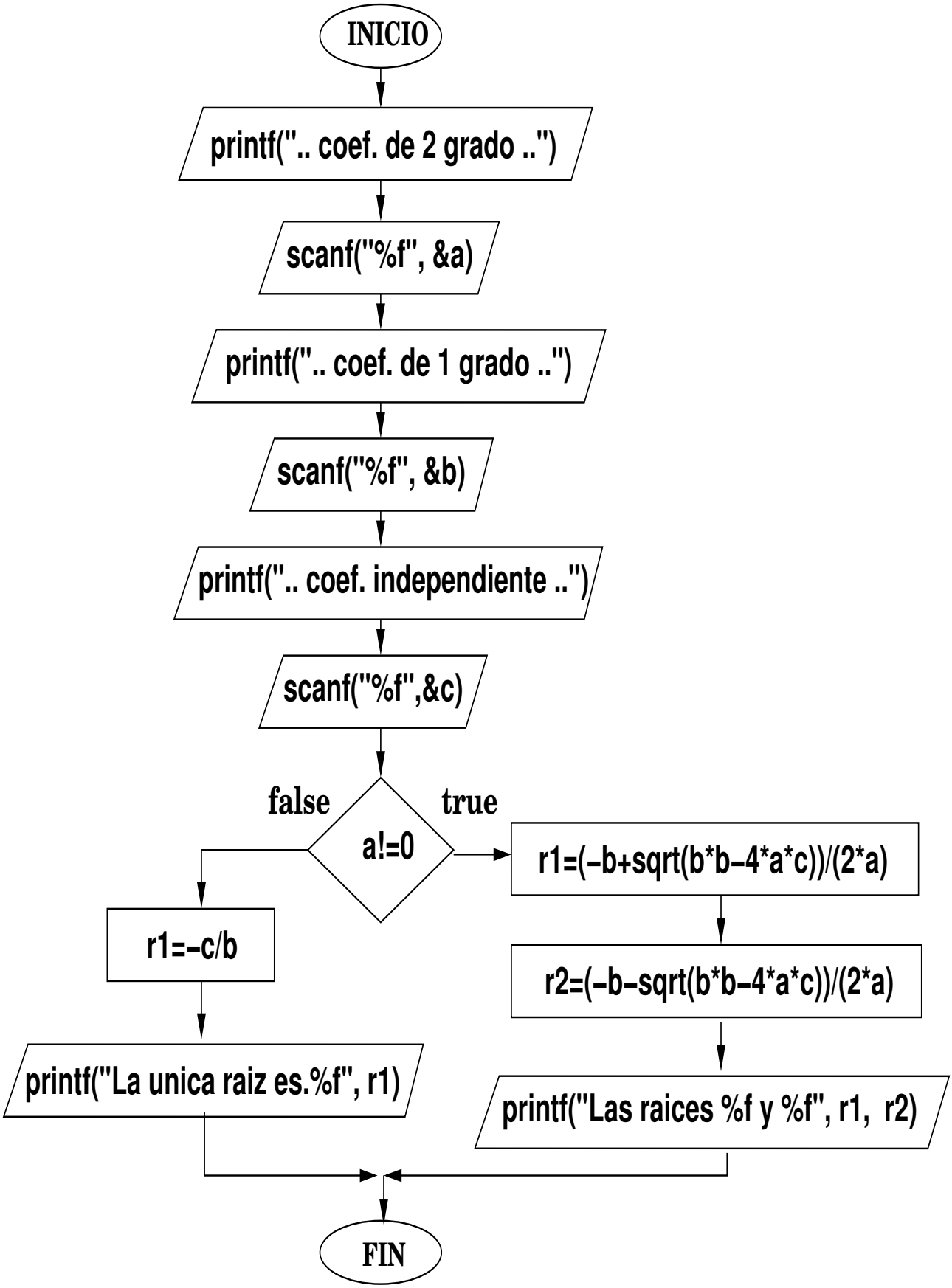
```
if (a!=0) {
   r1 = ( -b + sqrt( b*b-4*a*c ) ) / (2*a);
   r2 = ( -b - sqrt( b*b-4*a*c ) ) / (2*a);
   printf("Las raíces son %f y %f\n", r1, r2);

}

if (a==0)
   printf("Tiene una única raíz %f\n",-c/b);
```

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

2.1.4. CONDICIONAL DOBLE

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




INICIO
printf(".. coef. de 2 grado ..")
scanf("%f", &a)
printf(".. coef. de 1 grado ..")
scanf("%f", &b)
printf(".. coef. independiente ..")
scanf("%f",&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)
printf("Las raices %f y %f", r1, r2)
r1=–c/b
printf("La unica raiz es.%f", r1)
FIN

```
/* Programa que calcula las raíces de una ecuación
  de segundo grado (TERCERA APROXIMACIÓN)
*/
#include <stdio.h>
#include <math.h>


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

   printf("\nIntroduce coeficiente de 2º grado: ");
   scanf("%f", &a);
   printf("\nIntroduce coeficiente de 1er grado: ");
   scanf("%f", &b);
   printf("\nIntroduce coeficiente independiente: ");
   scanf("%f", &c);

   if (a!=0) {
      r1 = ( -b + sqrt( b*b-4*a*c ) ) / (2*a);
      r2 = ( -b - sqrt( b*b-4*a*c ) ) / (2*a);
      printf("Las raíces son %f y %f\n", r1, r2);
   }
   else {
      r1=-c/b;
      printf("La única raíz es %f\n", r1);
   }
}
```

Ejemplo. Calculad el mayor de dos valores `a` y `b`.

```
scanf("%d", &a);
scanf("%d", &b);

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

printf("El mayor es %d\n", max);
```

```
scanf("%d",&a);
scanf("%d",&b);

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

Línea en blanco

Tabulador
(recomendación: 3 espacios)

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

Compila correctamente **:-(**

PERO Suspenso automático **:-O**

Ejercicio. **En una variable `edad` guardamos la edad de una persona y en otra variable `ingresos`, sus ingresos. Subid sus ingresos en un `5%` si es un jubilado con unos ingresos inferiores a 300 euros e imprimid el resultado por pantalla. En caso contrario imprimid el mensaje `"No es aplicable la subida"`.**

Ejemplo. **El mayor de tres números `a`, `b` y `c` (primera aproximación).**

***Algoritmo*:**

▷ ***Entradas*: `a`, `b` y `c`**
***Salidas*: El mayor entre `a`, `b` y `c`**

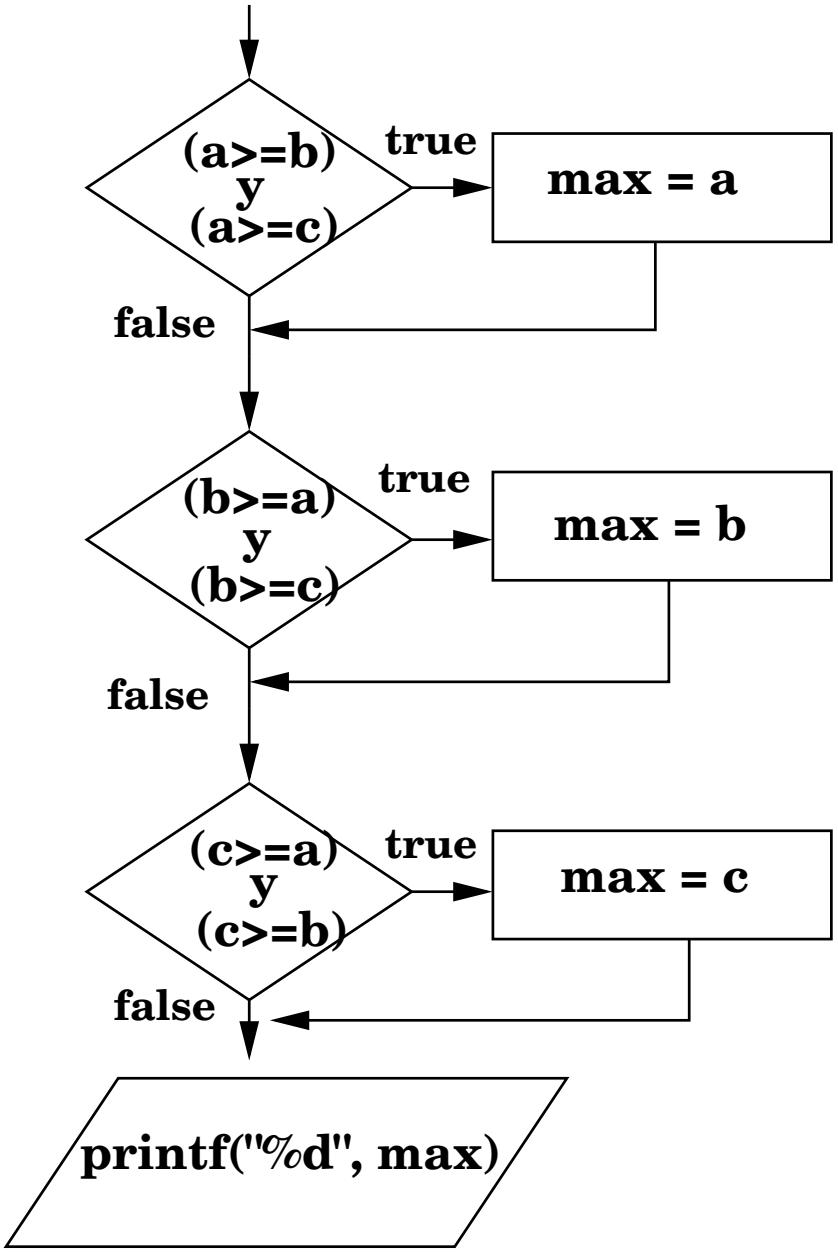
▷ ***Descripción*:**

Si `a` es mayor que los otros, el mayor es `a`

Si `b` es mayor que los otros, el mayor es `b`

Si `c` es mayor que los otros, el mayor es `c`

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

printf("El mayor es %d\n", max);
```


(a>=b)
y
(a>=c)
true
max = a
false
(b>=a)
y
(b>=c)
true
max = b
false
(c>=a)
y
(c>=b)
true
max = c
false
printf("%d", max)

2.1.5. 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)**
***Algoritmo*:**

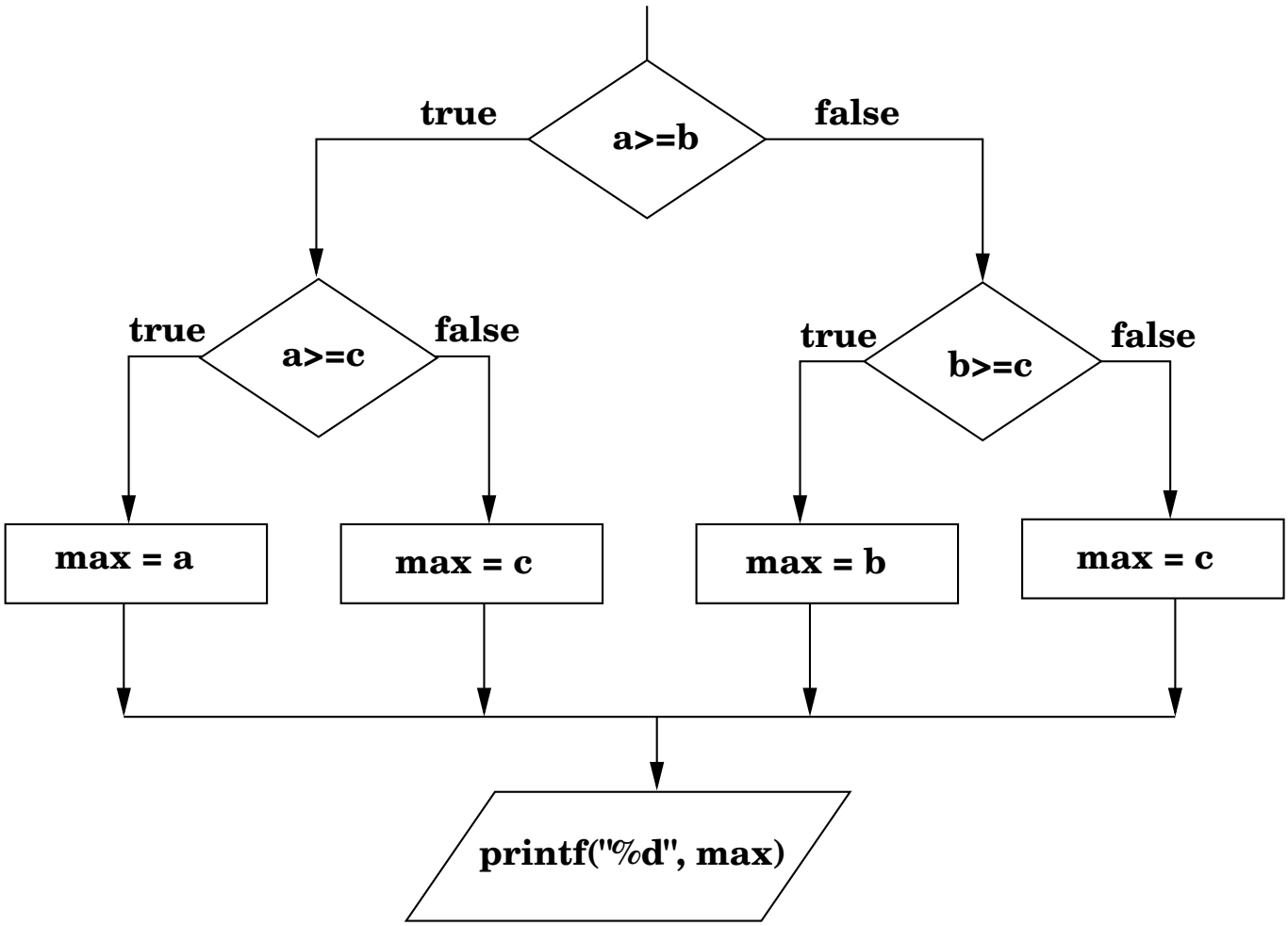
- ***Entradas y Salidas*: idem**
- ***Descripción*:**

  Si `a` es mayor que `b`, entonces:

  Calcular el mayor entre `a` y `c`

  En otro caso,

  Calcular el mayor entre `b` y `c`



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

printf("El mayor es %d\n", max);
```

Descripción de un algoritmo: Se trata de describir la idea principal del algoritmo, de forma concisa y esquemática, sin entrar en detalles innecesarios

Decripciones lamentables:

Compruebo si a>=b. En ese caso, lo que hago es ver si a>=c. En ese caso, basta asignarle a max el valor a. Si no es así, entonces le asigno c. Si no se verifica que a>=b, entonces compruebo si b>=c. Si es así, le asigno b y en caso contrario le asigno c.

El *colmo*:

Compruevo si a>=b y en ese caso lo que ago es ver si a>=c en ese caso basta asignarle a max el valor a si no es así entonces le asigno c si no se verifica que a>=b entonces compruevo si b>=c si es así le asigno b y en caso contrario le asigno c.

Inclusión de la descripción del algoritmo en el código:

- **Antes de un bloque**
- **Nunca entre las líneas del código**

```
/*
Si a es mayor que b, entonces:
   Calcular el mayor entre a y c
En otro caso,
   Calcular el mayor entre b y c
*/
if (a>=b)
   if (a>=c)
      max = a;
   else
      max = c;
else
   if (b>=c)
      max = b;
   else
      max = c;
```

También es válido incluir la descripción a la derecha del código, pero es mucho más difícil de mantener.

```
if (a>=b)
   if (a>=c)     /* Si a es mayor que b, entonces: */
      max = a;   /*  Calcular el mayor entre a y c */
   else          /* En otro caso,                  */
      max = c;   /*  Calcular el mayor entre b y c */
else
   if (b>=c)
      max = b;
   else
      max = c;
```

NUNCA han de mezclarse los comentarios con el código. Intentad leer este código `:-(`

```
/* Si a es mayor que b, entonces: */
if (a>=b)
   /* Calcular el mayor entre a y c */
   if (a>=c)
      max = a;
   else
      max = c;
/* En otro caso */
else
/* Calcular el mayor entre b y c */
   if (b>=c)
      max = b;
   else
      max = c;
```

No seguir las normas de codificación de programas baja puntos en el examen.

Ejemplo. El mayor de tres números (tercera aproximación)

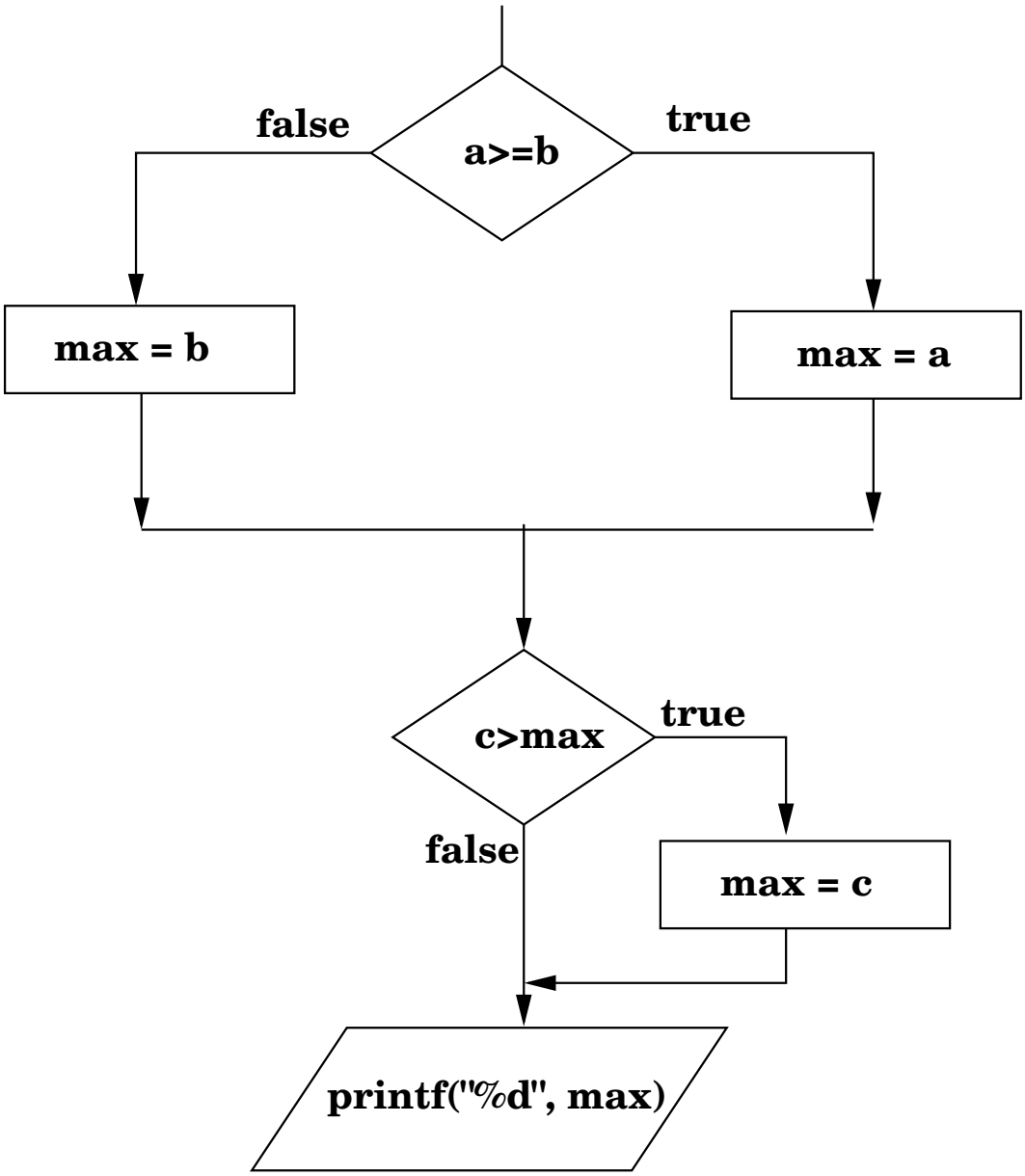
Algoritmo:

- ***Entradas y Salidas***: idem
- ***Descripción***:

  Calcular el máximo (`max`) entre `a` y `b`.

  Calcular el máximo entre `max` y `c`.

```
/*
   max = máximo entre a y b
   max = máximo entre max y c
*/

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

printf("El mayor es %d\n", max);
```


a>=b
false
true
max = b
max = a
c>max
true
false
max = c
printf("%d", max)

¿Con qué algoritmo 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 realiza alguna evaluación menos, pero la solución 3 es muchísimo mejor:

- **Es mucho más fácil de entender**
- **No repite código**
- **Es mucho más fácil de extender a varios valores.**

Máxima en la programación:
JAMÁS ha de repetirse código.
La repetición de código hace que los programas sean propensos a errores y sean más difíciles de actualizar y extender

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;

printf("El mayor es %d\n", max);
```

Aproximación 2:

Algoritmo:

- ▷ **Entradas y Salidas: idem**
- ▷ **Descripción:**

  Calcular el máximo (`max`) entre `a` y `b`.

  Calcular el máximo entre `max` y `c`.

  Calcular el máximo entre `max` y `d`.

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

printf("El mayor es %d\n", max);
```

En general:

Calcular el máximo (`max`) entre `a` y `b`.

Para cada uno de los valores restantes, calcular el máximo entre `max` y dicho valor.

Ejemplo: Estructuras condicionales anidadas

```
 /*
 Programa que permite sumar o restar dos números enteros
 */

#include <stdio.h>

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

  printf("\nIntroduce el primer operando: ");
  scanf("%d", &dato1);
  printf("\nIntroduce el segundo operando: ");
  scanf("%d", &dato2);
  printf("\nSelecciona (S)Sumar, (R)Restar, (M)Multiplicar: ");
  scanf("%c",&opcion);

  if (opcion == 'S')
     printf("Suma = %d\n", dato1+dato2);
  if (opcion == 'R')
     printf("Resta = %d\n", dato1-dato2);
  if (opcion == 'M')
     printf("Multiplicación = %d\n", dato1*dato2);
  if (opcion != 'R' && opcion != 'S' && opcion != 'M')
     printf("Ninguna operación");
}
```

Ejemplo: Estructuras condicionales anidadas (continuación)

Lo hacemos más eficientemente:

```
if (opcion == 'S')
  printf("Suma = %d\n", dato1+dato2);
else
    if (opcion == 'R')
       printf("Resta = %d\n", dato1-dato2);
    else
       if (opcion == 'M')
          printf("Multiplicación = %d\n", dato1*dato2);
       else
          printf("Ninguna operación");
```

Una forma también válida de tabular:

```
if (opcion == 'S')
   printf("Suma = %d\n", dato1+dato2);
else if (opcion == 'R')
   printf("Resta = %d\n", dato1-dato2);
else if (opcion == 'M')
   printf("Multiplicación = %d\n", dato1*dato2);
else
   printf("Ninguna operación");
```

2.1.6. CUESTIONES ADICIONALES

2.1.6.1. ¿ANIDAR O NO ANIDAR?

Un ejemplo en el que es mejor no anidar.

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 porque:

- **Al aumentar el anidamiento se va perdiendo en legibilidad y concisión.**
- **Muy importante: la segunda opción repite código (`bloque_B`) y por tanto es más propenso a errores de escritura o ante posibles cambios futuros.**

Un ejemplo en el que es mejor anidar.

Ejemplo. **Reescribir el siguiente ejemplo utilizando únicamente estructuras condicionales no anidadas:**

```
if (A>B)
     if (B<=C)
          if (C!=D)
               <S1>;
          else
               <S2>;
     else
          <S3>;
else
     <S4>;
```

```
if ((A>B) && (B<=C) && (C!=D))
   <S1>;
if ((A>B) && (B<=C) && (C==D))
   <S2>;
if ((A>B) && (B>C))
   <S3>;
if (A<=B)
   <S4>;
```

Preferimos el primero ya que realiza menos comprobaciones, y sobre todo porque no duplica código (el de las condiciones) y es por tanto, menos propenso a errores de escritura o ante posibles cambios futuros.

Ejercicio. **Para salir de una crisis económica, el gobierno decide:**

- **Bajar un 1 % el gravamen fiscal a los autónomos**
- **Para las rentas de trabajo:**
  - **Subir un 6 % el gravamen fiscal a las rentas de trabajo inferiores a 20.000 euros**
  - **Subir un 8 % el gravamen fiscal a los casados con rentas de trabajo superiores a 20.000 euros**
  - **Subir un 10 % el gravamen fiscal a los solteros con rentas de trabajo superiores a 20.000 euros**

```
if (es_autonomo)
   gravamen = 0.99 * gravamen;
else{
   if (renta<20000)
      gravamen = 1.06 * gravamen;
   else
      if (es_soltero)
         gravamen = 1.1 * gravamen;
      else
         gravamen = 1.08 * gravamen;
}
```

2.1.6.2. Simplificación de expresiones: Álgebra de Boole

Debemos intentar simplificar las expresiones lógicas, para que sean fáciles de entender. Usaremos las leyes del Álgebra de Boole, muy similares a las conocidas en Matemáticas elementales como:

$$-(a+b) = -a - b \qquad a(b+c) = ab + ac$$

```
!(A || B)        equivale a   !A && !B
!(A && B)        equivale a   !A || !B
A && (B || C)    equivale a   (A && B) || (A && C)
A || (B && C)    equivale a   (A || B) && (A || C)
```

***Nota*. Las dos primeras (relativas a la negación) se conocen como *Leyes de De Morgan*. Las dos siguientes son la ley distributiva (de la conjunción con respecto a la disyunción y viceversa). Para más información:**

```
http://es.wikipedia.org/wiki/Algebra_booleana
http://serbal.pntic.mec.es/~cmunoz11/boole.pdf
```

Ejemplo. **Es trabajador activo si su edad no está por debajo de 18 o por encima de 65.**

```
int es_activo;
int edad;

scanf("%d", &edad);
es_activo = !(edad<18 || edad>65);

if (es_activo)
   printf("\nTrabajador en activo");
......
```

Aplicando las leyes de De Morgan:

```
!(edad<18 || edad>65)
   equivale a
!(edad<18) && !(edad>65)
   equivale a
(edad >= 18) && (edad <= 65)
```

Es más fácil de entender el código:

```
es_activo = (edad >= 18) && (edad <= 65);

if (es_activo)
   printf("\nTrabajador en activo");
......
```

Ejemplo. **Es posible beneficiario de ayudas si, ya sea menor de 18 años o mayor de 65, la unidad familiar a la que pertenezca no puede tener una base imponible mayor de 35000 euros.**

```
bool beneficiario;
.......
beneficiario = ((base_imponible <= 35000) && (edad < 18))
               ||
               ((base_imponible <= 35000) && (edad > 65));
```

Es mucho más fácil de entender lo siguiente (aplicando la ley distributiva):

```
beneficiario = (base_imponible <= 35000) &&
               ( edad < 18 || edad > 65 );
```

Además, no repetimos código (`base_imponible <= 35000`**)**

2.1.6.3. PARTICULARIDADES DE C

El tipo `bool` como un tipo entero

En C, no hay el tipo lógico booleano se hace con un tipo entero. Cualquier expresión entera que devuelva el cero, se interpretará como `false`. Si devuelve cualquier valor distinto de cero, se interpretará como `true`.

```
int var_logica;

var_logica = 7;
var_logica = (4>5);  /* Correcto: resultado 0 false */
var_logica = 0;      /* Correcto: resultado 0 (false) */

var_logica = (4<5);  /* Correcto: resultado 1 true */
var_logica = 27;
var_logica = 2;      /* Correcto: resultado 2 (true) */
```

Esta dualidad nos puede dar quebraderos de cabeza:

```
int dato = 4;

if (! dato < 5)
   printf("%d es mayor o igual que 5", dato);
else
   printf("%d es menor de 5", dato);
```

El operador `!` **tiene más precedencia que** `<`**. Por lo tanto, la evaluación es como sigue:**

`! dato < 5` $\Leftrightarrow$ `(!dato) < 5` $\Leftrightarrow$ **(**`dato` **vale 4)** `(!true) < 5` $\Leftrightarrow$

$\Leftrightarrow$ `false < 5` $\Leftrightarrow$ `0 < 5` $\Leftrightarrow$ `true`

¡Imprime `4 es mayor o igual que 5`**!**

Solución:

```
if (! (dato < 5))
   printf("%d es mayor o igual que 5", dato);
else
   printf("%d es menor de 5", dato);
```

o mejor:

```
if (dato >= 5)
   printf("%d es mayor o igual que 5", dato);
else
   printf("%d es menor de 5", dato);
```

***Consejo*: No hay que ser rácano con los paréntesis**

El operador de asignación en expresiones

El operador de asignación `=` se usa en sentencias del tipo:

```
valor = 7;
```

Pero además, devuelve un valor: el resultado de la asignación. Así pues, `valor = 7` *es una expresión* que devuelve `7`

```
un_valor = otro_valor = valor = 7;
```

Esto producirá fuertes dolores de cabeza cuando por error usemos una expresión de asignación en un condicional:

```
valor = 5;

if (valor = 7)
   <acciones if>   /* Siempre se ejecuta este bloque */
else
   <acciones else>

/* Además, valor se queda con 7 */
```

`valor = 7` devuelve `7`. Al ser distinto de cero, es `true`. Por tanto, se ejecuta el bloque `if` (y además `valor` se ha modificado con `7`)

```
a = 7;

if (a=0)
   printf("\nRaíz= %d", -c/b);  /* Nunca se ejecuta */
else{
   r1 = -b + sqrt(b*b -4*a*c) / (2*a) ;  /* Error en ejecución *
   .......
}
```

El operador de incremento en expresiones

```
variable = 9;

if (variable+1 == 10)
   printf("%d es igual a 9", variable);   /* Correcto */

printf("%d", variable);   /* Imprime 9 */
```

El operador ++ (y --) forma una expresión: devuelve el valor incrementado:

```
variable = 9;

if (variable++ == 10)   /* No entra pues el ++ esta despues*/
   printf("%d es igual a 9", variable); /* 10 es igual a 9 */

printf("%d", variable);  /* 10 */
```

***Consejo*: No usad = ni ++ en la expresión lógica de una sentencia condicional**

2.1.6.4. 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.**

Por lo tanto, pondremos primero aquellas condiciones que sean más probables de evaluarse como `false`

***Nota*. La misma idea pasa cuando el operador es** `||` **pero se ponen primero aquellas condiciones que sean más probable evaluarse como** `true`**.**

2.1.6.5. CUIDADO CON LA COMPARACIÓN ENTRE REALES

La expresión `1.0 == (1.0/3.0)*3.0` podría evaluarse a `false` debido a la precisión finita para calcular `(1.0/3.0)` (`0.333333333`)

Es más:

```
float descuento_base, porcentaje;

descuento_base = 0.1;
porcentaje = descuento_base * 100;

if (porcentaje == 10)   /* Podría evaluarse a false */
   printf("Descuento del 10%");
....
```

Recordad que la representación en coma flotante no es precisa y 0.1 será internamente un valor próximo a 0.1, pero no exacto.

***Soluciones*:**

- ▷ **Fomentar condiciones de desigualdad**
- ▷ **Mejor: Fijar un *error* de precisión y aceptar igualdad cuando la diferencia de las cantidades sea menor que dicho error.**

```
float real1, real2;
const float epsilon = 0.00001;

if ((real1-real2) < epsilon)
   printf("Son iguales");
```

Como no es lo mismo comparar si `0.000001` **está cerca de** `0.000001`**, que comparar si** `130.001` **está cerca de** `130.002`**, el método anterior hay que refinarlo. Para más información:**

`http://www.codeguru.com/forum/showthread.php?t=323835`

`http://www.cygnus-software.com:80/papers/`
`/comparingfloats/comparingfloats.htm`

2.1.7. Estructura Condicional Múltiple

Recordemos el ejemplo de lectura de dos enteros y una opción:

```
int dato1, dato2;
char opcion;

printf("\nIntroduce el primer operando: ");
scanf("%d", &dato1);
printf("\nIntroduce el segundo operando: ");
scanf("%d", &dato2);
printf("\nSelecciona (S) sumar, (R) restar, (M) multiplicar: "
scanf("%c", &opcion);

if (opcion == 'S')
   printf("Suma = %d\n", dato1+dato2);
else if (opcion == 'R')
   printf("Resta = %d\n", dato1-dato2);
else if (opcion == 'M')
   printf("Multiplicación = %d\n", dato1*dato2);
else
   printf("Ninguna operación\n");
```

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

- ▷ **<*expresión*> es un expresión entera o char.**
- ▷ **<*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
*/
#include <stdio.h>

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

  printf("\nIntroduce el primer operando: ");
  scanf("%d", &dato1);
  printf("\nIntroduce el segundo operando: ");
  scanf("%d", &dato2);
  printf("\nSelecciona (S) sumar, (R) restar: ");
  scanf("%c", &opcion);

  switch (opcion) {
     case 'S': printf("Suma = %d\n", dato1+dato2);
               break;
     case 'R': printf("Resta = %d\n", dato1-dato2);
               break;
     case 'M': printf("Multiplicación = %d\n", dato1*dato2);
               break;
     default:
               printf("Ninguna operación\n");
  }
}
```

El gran problema con la estructura `switch` es que el programador olvidará en más de una ocasión, incluir la sentencia `break`. La única *ventaja* es que se pueden realizar las mismas operaciones para un grupo determinado de constantes:

```
switch (opcion) {
  case 's':
  case 'S': printf("Suma = %d\n", dato1+dato2);
            break;
  case 'r':
  case 'R': printf("Resta = %d\n", dato1-dato2);
            break;
  case 'm':
  case 'M': printf("Multiplicación = %d\n", dato1*dato2);
            break;
  default:  printf("Ninguna operación\n");
            break;
            /* Ponemos break; aunque no sea necesario */

}
```

Pero para conseguir ese objetivo, la siguiente solución sería mucho mejor:

```
opcion = toupper(opcion);

switch (opcion) {
  case 'S': printf("Suma = %d\n", dato1+dato2);
            break;
  case 'R': printf("Resta = %d\n", dato1-dato2);
            break;
  case 'M': printf("Multiplicación = %d\n", dato1*dato2);
            break;
  default:
            printf("Ninguna operación\n");
            break;
            /* Ponemos break; aunque no sea necesario */
}
```

Y para no tener errores lógicos si se nos olvida incluir `break`**, podemos volver a lo que ya sabemos:**

```
opcion = toupper(opcion);

if (opcion == 'S')
   printf("Suma = %d\n", dato1+dato2);
else if (opcion == 'R')
   printf("Resta = %d\n", dato1-dato2);
else if (opcion == 'M')
   printf("Multiplicación = \n", dato1*dato2);
else
   printf("Ninguna operación\n");
```

> ***Consejo*: En la medida de lo posible, evitad el uso de `switch` por los errores lógicos producidos al olvidarnos incluir algún `break`**

2.2. Estructuras Repetitivas

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)**

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

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

2.2.1.1. FORMATO

Pre-test

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

Post-test

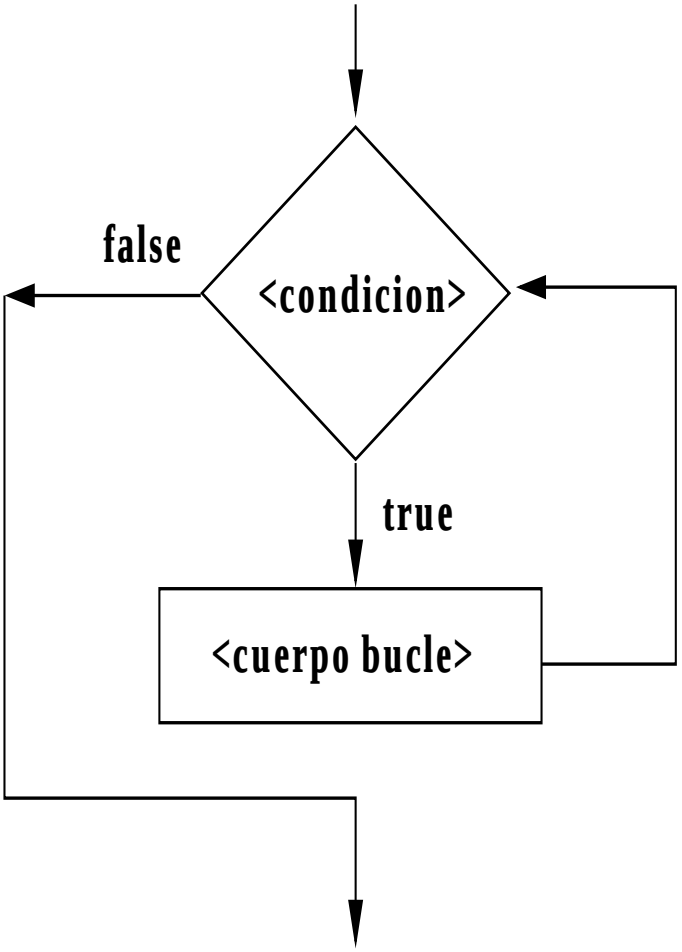
```
do {
   <cuerpo bucle>
} while (<condición>);
```

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

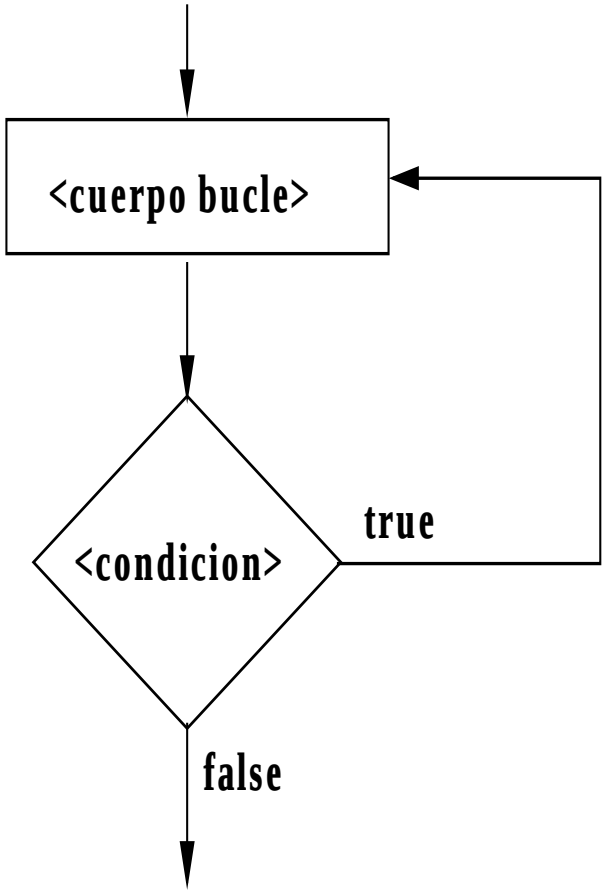
- ▷ **En un bucle `while`, primero se pregunta y luego (en su caso) se ejecuta.**
- ▷ **En un bucle `do while`, primero se ejecuta y luego se pregunta.**

Cada vez que se ejecuta el cuerpo del bucle diremos que se ha producido una *iteración*

(Pre-test)



(Post-test)



2.2.1.2. ALGUNOS USOS DE LOS BUCLES

Ejemplo. **Crear un *filtro* de entrada de datos: Leer un valor y no permitir al usuario que lo introduzca fuera de un rango determinado.**

```
/* Introducir un único valor, pero positivo.
   Imprimir el coseno. */

#include <stdio.h>
#include <math.h>


int main(){
  int valor;

  do{
     printf("\nIntroduzca un valor positivo: ");
     scanf("%d", &valor);
  }while (valor < 0);

  printf("%f", cos(valor));
}
```

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{
   printf("*****\n");
   total = total + 1;        /* nuevo = antiguo + 1 */
}while (total <= 20);
```

O bien:

```
total = 0;

do{
   printf("*****\n");
   total = total + 1;
}while (total < 20);
```

O bien:

```
total = 1;

while (total <= 20){
   printf("*****\n");
   total = total + 1;
}
```

Preferible:

```
total = 0;   /* Llevo 0 líneas impresas */

while (total < 20){
   printf("*****\n");
   total = total + 1;
}
```

Dentro del bucle, JUSTO antes de comprobar la condición, la variable `total` contiene el número de líneas que hay impresas `:-)`

***Nota*. Podríamos usar el operador de incremento:**

```
while (total < 20){
   printf("*****\n");
   total++;
}
```

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

```
printf("\n¿Cuántas líneas de asteriscos quiere imprimir? ");
scanf("%d", &tope);
total = 0;

do{
   printf("*****\n");
   total++;
}while (total < tope);
```

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

```
scanf("%d", &tope);
total = 0;

while (total < tope){
   printf("*****\n");
   total++;
}
```

Ejemplo. **Leed un entero `tope` y escribir los pares $\leq$ `tope`**

```
scanf("%d", &tope);
par = 0;

while (par <= tope){
   par = par+2;
   printf("%d", par);
}
```

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

```
scanf("%d", &tope);
par = 0;                  /* Primer candidato */

while (par <= tope){      /* ¿Es bueno? */
   printf("%d", par);     /* Si => imprímelo */
   par = par+2;           /*       calcular nuevo */
                          /*       candidato */
}                         /* No => Salir */
```

Si no queremos que salga 0, cambiaríamos la *inicialización*:

```
par = 2
```

Ejercicio. **Dado un entero `n`, calculad el número de dígitos que tiene.**

```
57102   ->   5 dígitos
45      ->   2 dígitos
```

***Algoritmo*:**

▷ ***Entradas*:** `n`

▷ ***Salidas*:** `num_digitos`

▷ ***Descripción*:**

```
Ir dividiendo n por 10 hasta llegar a 1 cifra
El número de dígitos será el número de iteraciones
```

```
num_digitos = 1;

while (n > 9){
    n = n/10;
    num_digitos++;
 }
```

2.2.1.3. LECTURA ANTICIPADA

Ejemplo.

```
/* Programa que suma los valores leídos con scanf
   hasta que se lea el valor -1 (terminador = -1) */
#include<stdio.h>

int main(){
  int suma, numero;

  suma = 0;

  do{
     scanf("%d", &numero);
     suma = suma + numero;
  }while (numero != -1);

  printf("La suma es %d\n", suma);
}
```

***Problema*: Procesa el `-1` y lo suma.**

***Solución*:**

```
  do{
     scanf("%d", &numero);
     if (numero != -1)
        suma = suma + numero;
  }while (numero != -1);
```

Funciona, pero evalúa dos veces la misma condición.

Recordad: Evitad el uso de bucles `do while`

***Solución*: técnica de *lectura anticipada*. Leemos el primer valor con** `scanf` **antes de entrar al bucle y comprobamos si hay que procesarlo (el primer valor podría ser ya el terminador!)**

```
  suma = 0;
  scanf("%d", &numero);          /* Primer candidato */

  while (numero != -1) {         /* ¿Es bueno? */
     suma = suma + numero;
     scanf("%d", &numero);       /* Leer siguiente candidato */
  }

  printf("La suma es %d\n", suma);
}
```

***Nota*. La primera vez que entra al bucle, la instrucción** `while (numero != -1)` **hace las veces de un condicional.**

***Nota*. Somos conscientes de que se repite cierto código:**

```
   scanf("%d", &numero);
```

Lo arreglaremos cuando veamos las funciones.

Esta lectura anticipada nos permite ser más específicos con los mensajes en pantalla:

```
#include <stdio.h>

int main(){
  int suma, numero;

  suma = 0;

  printf("\nIntroduzca un valor entero\n");
  printf("Para terminar introduzca -1");
  printf("\nValor: ");
  scanf("%d", &numero);      /* Primer candidato */

  while (numero != -1) {     /* ¿Es bueno? */
     suma = suma + numero;   /* Lo procesamos */
     printf("\nValor: ");
     scanf("%d", &numero);   /* Leemos siguiente candidato */
  }

  printf("La suma es %d\n", suma);
}
```

Otro ejemplo de lectura anticipada:

```
/* Programa que va leyendo números enteros hasta que se
   lea el cero. Imprimir el número de pares e impares leídos. */

#include <stdio.h>


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

   ContPar=0;
   ContImpar=0;
   printf("Introduce valor: ");
   scanf("%d", &valor);

   while (          ) {
      if (          )
                                  ;
      else
                                  ;

      printf("Introduce valor: ");
      scanf("%d", &valor);
   }

   printf("Fueron %d pares y %d impares\n",ContPar,ContImpar);
}
```

```
scanf("%d", &valor);
```

← Línea en blanco antes

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

    printf("Introduce ...");
    scanf("%d", &valor);
}
```

← Línea en blanco después

```
printf("Fueron " ....);
```

2.2.1.4. CONDICIONES COMPUESTAS

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

```
char opcion;
do{
   printf("¿Desea formatear el disco?");
   scanf("%c", &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.**

Ejercicio. **Calcular el máximo común divisor de dos números `a` y `b`. Probad con todos los enteros menores que el menor de ambos números, hasta llegar al primero que divida a los dos.**

***Algoritmo*:**

- ***Entradas*: Los dos enteros `a` y `b`**
  ***Salidas*: el entero MCD de `a` y `b`**
- ***Descripción*:**
  - **Calcular el `menor` entre `a` y `b`**
  - **Probar con todos los enteros menores que `menor`, hasta encontrar uno que divida a ambos (`a` y `b`).**

Ejercicio. **Encontrar el primer divisor múltiplo de 7 de un entero `n`. El divisor ha de ser propio (distinto de 1 y `n`)**

```
int n, divisor, tope;
scanf("%d", &n);     divisor = 7;    tope = n/2;

while (divisor<=tope  &&  n%divisor != 0)
   divisor = divisor+7;

/* Ahora Debemos comprobar qué condición fue falsa */

if (n%divisor == 0)
   printf("El primer divisor de %d múltiplo de 7 es: %d\n",
                                                n, divisor) ;
```

Repetimos código: `n%divisor==0` **es equivalente a** `n%divisor!=0`

```
int es_menor, encontrado;
scanf("%d", &n);     divisor = 7;     tope = n/2;
encontrado = 0;      es_menor = 1;

while (es_menor && !encontrado){
   if (divisor > tope)
      es_menor = 0;
   else if (n % divisor == 0)
      encontrado = 1;
   else
      divisor = divisor +7;
}
if (encontrado)
   printf("%d", divisor);
```

Ejemplo. **Leer números enteros de la entrada por defecto hasta que se introduzcan 10 ó hasta que se introduzca un número negativo. Imprimir la media aritmética.**

```
suma = 0;
scanf("%d", &valor);
contador = 1;

while ((valor >= 0) && (contador<=10)){
   suma = suma + valor;
   contador++;
   scanf("%d", &valor);
}
media = suma/contador;
```

Problema**: Lee el undécimo y se sale, pero ha tenido que leer dicho valor.**
Solución**: Leemos hasta 9 y el décimo lo sumamos al final.**

```
suma = 0;
scanf("%d", &valor);
contador = 1;

while ((valor >= 0) && (contador<10)){
   suma = suma + valor;
   contador++;
   scanf("%d", &valor);
}
suma = suma + valor;
media = suma/contador;
```

***Problema*: Si se ha salido con un negativo lo suma. Además, `contador` también se incrementa en uno con el valor negativo, por lo que habría que tenerlo en cuenta a la hora de hallar la media aritmética. Quedaría:**

```
suma = 0;
scanf("%d", &valor);
contador = 1;

while ((valor >= 0) && (contador<10)){
   suma = suma + valor;
   contador++;
   scanf("%d", &valor);
}

if (contador == 10)
   suma = suma + valor;
else
   contador--;

media = suma/contador;
```

¿Pero y si el último leído es negativo?

```
if ((contador == 10) && (valor >=0))
   suma = suma + valor;
else
   contador--;
```

Podemos observar la complejidad (por no decir chapucería) innecesaria que ha alcanzado el programa. En estos casos, es mejor replantear desde el principio la solución y si tenemos

una expresión lógica complicada, en la que las condiciones interactúan entre sí, usamos mejor una variable lógica en la forma siguiente:

```
suma = 0;
contador = 0;
seguir = 1;

while (seguir){   /* seguir-> (valor>0) y (<= 10 valores) */
   scanf("%d", &valor);

   if (valor < 0)
      seguir = 0;
   else{
      suma = suma + valor;
      contador++;

      if (contador == 10)
         seguir = 0;
   }
}
media = suma/contador;

if (contador == 0)  /* Ningún positivo introducido */
   printf("\nNo se introdujeron valores");
else
   printf("\nMedia aritmética = %d\n", media);
```

2.2.1.5. BUCLES SIN FÍN

Ejercicio. **¿Cuantas iteraciones se producen?**

```
contador = 2;

while (contador < 3) {
   contador--;
   printf("%d\n", contador);
}
```

```
contador = 1;

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

Fomentar el uso de condiciones de desigualdad.

```
contador = 1;

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

2.2.2. BUCLES `for`

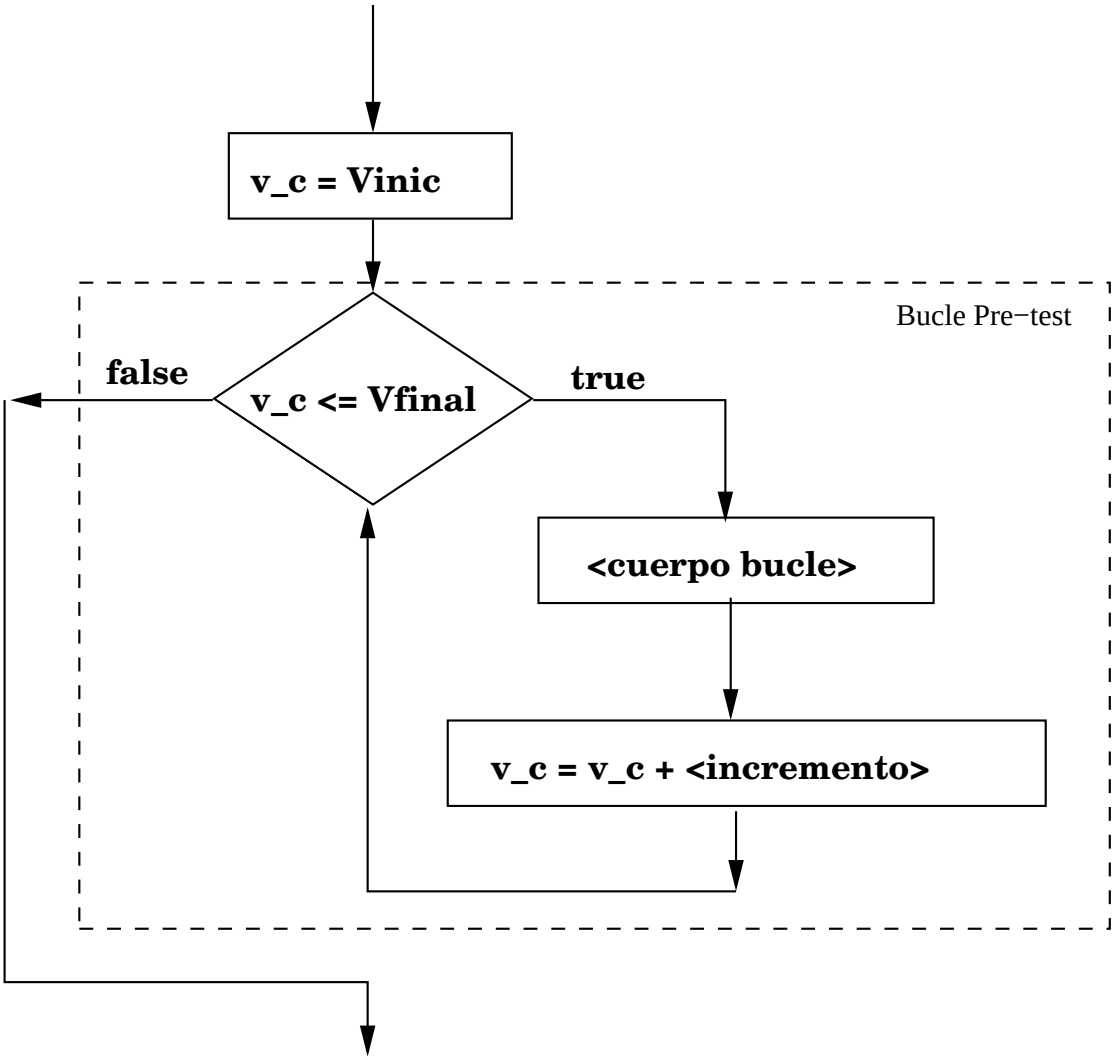
2.2.2.1. MOTIVACIÓN

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

Ejemplo. **Hallar la media aritmética de cinco enteros leídos desde teclado.**

```
int main(){
   int contador, valor, suma, inicio, final;
   float media;

   inicio = 1;
   final = 5;
   suma = 0;
   contador = inicio;                  /* v_c =  Vinic */

   while (contador <= final){          /* v_c <= Vfinal */
      printf("Introduce el número %dº: \n", contador);
      scanf("%d", &valor);
      suma = suma + valor;

      contador++;                      /* v_c = v_c + 1 */
   }
   media = suma / (final *1.0);
   printf("La media es %f\n", media);
}
```

Diagrama de flujo



`v_c` **es la variable controladora del ciclo.**

`Vinic` **valor inicial que toma** `v_c`**.**

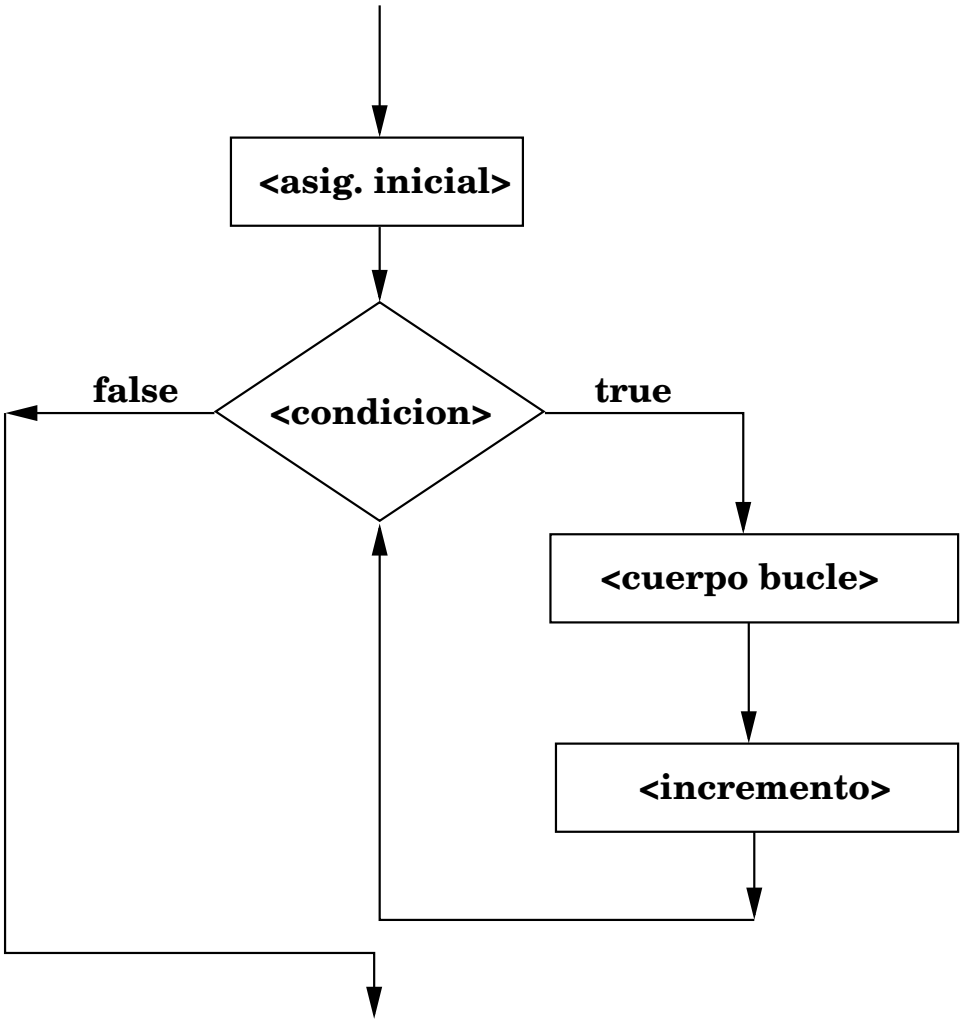
`Vfinal` **valor final que debe tomar** `v_c`**.**

2.2.2.2. 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 <stdio.h>

int main(){
   int contador, valor, suma, inicio, final;
   float media;

   inicio = 1;
   final = 5;
   suma = 0;

   for (contador=inicio ; contador <= final ; contador++){
      printf("Introduce el número %dº: \n", contador);
      scanf("%d", &valor);
      suma = suma + valor;
   }

   media = suma / (final*1.0);
   printf("La media es %f\n", media);
}
```

`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**

Encuentre errores en este código:

```
For {x = 100, x>=1, x++}
   printf("%d\n", x);
```

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

```
int candidato, num_pares;

num_pares = 0;

for(candidato = -10; candidato <= 10; candidato++) {
   if (candidato % 2 == 0)
      num_pares++;
}
printf("%d\n", num_pares);
```

¿Qué hace `num_pares++`?
En cada iteración se usa el valor de `num_pares` de la iteración anterior y se le suma `1`.

Este problema también se podría haber resuelto como sigue:

```
int num_pares, par;

for (par = -10; par <= 10; par = par+2)
   num_pares++;

printf("%d\", num_pares);
```

Ejemplo. Imprimir 9 veces el mensaje `Hola`

```
int i;

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

```
for (i = 9; i > 0; i--)
   printf("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)/incremento
```

Y el número de iteraciones será

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

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

Ejercicio. **¿Qué salida producen los siguientes trozos de código?**

```
int i, j, total;
total = 0;

for (i=1 ; i<=10; i++)
   total = total + 3;
printf("%d\n", total);
```

```
total = 0;

for (i=4 ; i<=36 ; i=i+4)
   total++;
printf("%d\n", total);
```

```
total = 0;
i = 4;

while (i <= 36){
   total++;
   i = i+4;
}
printf("%d\n", total);
```

2.2.3. EL BUCLE `for` EN C

2.2.3.1. 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**

C no impone dicha restricción, pero nosotros la adoptaremos como buenos programadores.

Ejemplo. **Sumar pares hasta un tope**

```
suma  = 0;

for (par=2; par<=tope ; par++){
   if (par%2 == 0)
      suma = suma+par;
   else
      par++;    /* Error lógico */
}
```

Escribe los enteros entre `inf` y `sup`, hasta llegar al primer múltiplo de 13.

```
scanf("%d", &inf);
scanf("%d", &sup);

for (entero = inf; entero <= sup; entero++) {
   printf("%d\n", entero);

   if (entero % 13 == 0)
      entero = sup+1;        /* Mal hábito */
}
```

Ejercicio. **Haced el problema anterior con un bucle `while`**

2.2.3.2. ALGUNAS CONSIDERACIONES SOBRE EL BUCLE `for`

▶ *Bucles* `for` *con cuerpo vacío*

¿Qué ocurriría en el siguiente código?

```
for (x = 1; x <= 20; x++);
  printf("%d\n", x);
```

Realmente, el código bien tabulado es:

```
for (x = 1; x <= 20; x++)
    ;
printf("%d\n", x);
```

A veces sí se usan bucles for con sentencias vacías: todo se hace dentro del `for`.

Ejemplo. **MCD de dos enteros `a` y `b`.**

```
if (b<a)
  menor = b;
else
  menor = a;

divisor = menor;

while ((a % divisor != 0) || (b % divisor != 0))
    divisor --;

MCD = divisor;
```

```
for (divisor = menor ; (a%divisor!=0) || (b%divisor!=0) ; divisor--)
    ;
MCD = divisor;
```

> ***Consejo***: **No abusad de este estilo de codificación, usando cuerpos de un bucle for sin ninguna sentencia**

► *Bucles* `for` *con sentencias de incremento incorrectas*

Lo siguiente es un error lógico:

```
int num_pares, par;

for (par = -10; par <= 10; par+2) /* en vez de par = par+2 */
   num_pares++;
```

equivale a:

```
par = -10;

while (par <= 10){
   num_pares++;
   par+2; /* Compila correctamente pero no incrementa par */
}
```

Resultado: bucle infinito.

2.2.3.3. 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>];
     [<expresión lógica>];
     [<sentencia final>])
    <cuerpo bucle>
```

donde,

- ▷ *<sentencia inicial>* **es la sentencia que se ejecuta antes de entrar al bucle,**
- ▷ *<expresión booleana>* **es cualquier condición que verifica si el ciclo debe terminar o no,**
- ▷ *<sentencia final>* **es la sentencia que se ejecuta antes de volver arriba para comprobar el valor de la expresión lógica.**

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. **Escribe los enteros entre `inf` y `sup`, hasta llegar al primer múltiplo de 13.**

```
scanf("%d", &inf);
scanf("%d", &sup);

entero = inf;

while(entero<=sup && entero%13 != 0){
   printf("%d\n", entero);
   entero++;
}
```

```
scanf("%d", &inf);
scanf("%d", &sup);

for (entero = inf; entero<=sup && entero%13 != 0 ; entero++)
   printf("%d\n", entero);
```

Ejemplo. Comprobar si un número es primo.

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

int main(){
   int valor, divisor;
   int es_primo;
   printf("Introduzca un numero natural: ");
   scanf("%d", &valor);

   es_primo = 1;

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

   if (es_primo)
      printf("%d es primo\n", valor);
   else
      printf("%d no es primo\n", valor);
}
```

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

```
es_primo = 1;

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

Incluso podríamos quedarnos en `sqrt(valor)`, ya que si `valor` no es primo, tiene al menos un divisor menor que `sqrt(valor)`

```
int valor, divisor;
int es_primo;
float tope;

printf("Introduzca un numero natural: ");
scanf("%d", &valor);

es_primo = 1;
tope = sqrt(valor);

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

if (es_primo)
   printf("%d es primo\n", valor);
else
   printf("%d no es primo\n", valor);
}
```

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

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

Ampliación:

Cuando `valor` es un entero muy grande, el algoritmo anterior no termina en un *tiempo razonable*.

▷ **¿Qué es un número grande?**

`http://www.naturalnumbers.org/bignum.html`

▷ **¿Qué significa *tiempo razonable*?**

Está fuera del alcance del curso e implica nociones más profundas sobre complejidad computacional

Para una introducción informal pero clara:

`http://www.claymath.org/Popular_Lectures/Minesweeper/`

El primer algoritmo para determinar en un tiempo razonable si un número es primo es bastante reciente (2002):

`Wikipedia -> AKS primality test`

▷ **Sobre números primos en general:**

`http://primes.utm.edu/`

En los dos ejemplos anteriores

```
for (entero = inf; entero<=sup && entero%13 != 0 ; entero++)
for (divisor = 2 ;
   divisor <= tope && es_primo ;
   divisor++)
```

se ha usado dentro del `for` una variable contadora y otra condición adicional. Esto es un uso habitual y correcto del bucle `for`.

¿Puede usarse entonces, cualquier condición dentro de la cabecera del `for`? Sí, pero no es muy recomendable.

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 <stdio.h>

int main(){
  int num_valores, valor;

  scanf("%d", &valor);
  num_valores = 0;           /* inicialización */

  while (valor != 0){        /* condición */
     scanf("%d", &valor);
     num_valores++;          /* incremento */
  }

  printf("El número de valores introduci. es %d",num_valores);
}
```

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

#include <stdio.h>



int main(){
  int  num_valores, valor;

  scanf("%d", &valor);

  for (num_valores=0; valor!=0; num_valores++)
     scanf("%d", &valor);

  printf("El número de valores introduci. es %d\n",num_valores);
}
```

Debemos evitar este tipo de bucles `for` en los que la(s) variable(s) que aparece en la condición, no aparece en las otras dos expresiones.

En el anterior ejemplo, si cambiamos los nombres de las variables, tendríamos:

```
for (recorr=0; recoger!=0; recorr++)
   .....
```

Y el cerebro de muchos programadores le engañará y le harán creer que está viendo lo siguiente, que es a lo que está acostumbrado:

```
for (recorr=0; recorr!=0; recorr++)   :-(
   .....
```

Esto no quiere decir que, ADEMÁS de la variable contadora, no podamos usar condiciones adicionales para salirnos del bucle. Será algo usual (lo hicimos en el ejemplo del primo)

Y ya puestos, ¿podemos suprimir algunas expresiones de la cabecera de un bucle `for`? La respuesta es que sí, pero hay que evitarlas SIEMPRE. Oscurecen el código.

Ejemplo. **Sumar valores leídos desde la entrada por defecto, hasta introducir un cero.**

```
int main(){
  int valor;
  int suma=0;

  scanf("%d", &valor);

  for ( ; valor!=0 ; ){      /* :-( */
     suma = suma + valor;
     scanf("%d", &valor);
  }
```

En resumen, ¿Cuando usar un ciclo `for`?

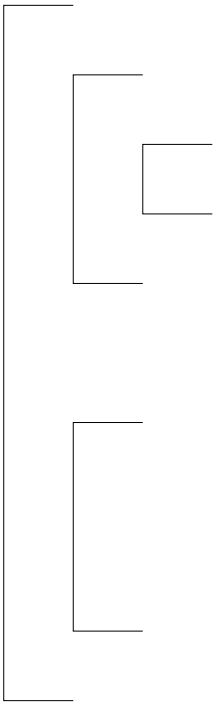
- **Por supuesto, en los ciclos controlados por contador**
- **Cuando el flujo de control del ciclo tenga SIEMPRE:**
  - **Una sentencia de inicialización del contador**
  - **Una condición de continuación que involucre al contador**
  - **Una sentencia de actualización que involucre al contador**

2.2.4. ANIDAMIENTO DE BUCLES

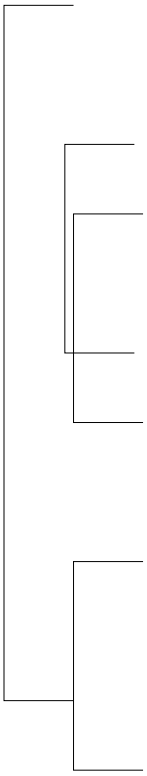
Dos bucles se encuentran anidados, cuando uno de ellos está en el bloque de sentencias del otro.

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 hasta N
   con los N primeros números. */

#include <stdio.h>

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

   for (i=1 ; i<=N ; i++) {
      for (j=1 ; j<=N ; j++)
         printf("%d * %d = %d", i, j, i*j);
      printf("\n");
  }
}
```

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<4; i++)
   for (j=0; j<3 ; j++)
      iteraciones++;
printf("%d\n", iteraciones);
```

```
iteraciones=0;

for (i=0; i<4; i++)
   for (j=i; j<3 ; j++)
      iteraciones++;
printf("%d\n", iteraciones);
```

```
 total = 3;

 for (i=2 ; i<=10 ; i=i+2)
     for (j=1 ; j<=8 ; j++)
         total++;
 printf("%d\n", total);
```

Ejemplo. **Imprimir en pantalla los divisores primos del entero** `N`

***Algoritmo*:**

▷ ***Entradas*: `N`**
***Salidas*: ninguna**

▷ ***Descripción*:**

Recorrer todos los enteros menores que `N`
Comprobar si el entero es primo. En dicho caso, comprobar si divide a `N`

Ineficiente. Es mejor el siguiente:

Recorrer todos los enteros menores que `N`
Comprobar si el entero divide a `N`. En dicho caso, comprobar si el entero es primo.

***Nota*. Mejor aún si empezamos desde `N/2`**

```
for (divisor_N = N/2 ; divisor_N > 1 ; divisor_N--){
   if (N % divisor_N == 0){
      es_primo = 1;
      tope = sqrt(1.0*divisor_N);

      for ( divisor_primo = 2 ;
            divisor_primo <= tope && es_primo ;
            divisor_primo++){

         if (divisor_N % divisor_primo == 0)
            es_primo = 0;
      }

      if (es_primo)
         printf("\nEl primo %d divide a %d", divisor_N, N);
   }
}
```

Ejemplo. **Imprimir en pantalla la descomposición en números primos de un valor entero.**

```
n     p
360   2
180   2
90    2
45    3
15    3
5     5
1
```

Fijamos un valor de `p` cualquiera. Por ejemplo `p=2`

```
Dividir n por p cuantas veces sea posible
```

```
p = 2;

while (n%p == 0){
   printf("%d ", p);
   n = n/p;
}
```

Ahora debemos pasar al siguiente primo `p` y volver a ejecutar el bloque anterior.

Condición de parada: `n>=p` o bien `n>1`

```
Mientras n>1
   Dividir n por p cuantas veces sea posible
   p = siguiente primo mayor que p
```

¿Cómo pasamos al siguiente primo?

```
p++;
Recorrer todos los enteros menores de n
hasta encontrar un divisor o hasta llegar a n
```

Este recorrido implica un bucle, pero no es necesario. Hagamos simplemente `p++`**:**

```
Mientras n>1
   Dividir n por p cuantas veces sea posible
   p++
```

```
p = 2;

while (n>1){
   while (n%p == 0){
      printf("%d ", p);
      n = n/p;
   }
   p++;
}
```

¿Corremos el peligro de intentar dividir `n` **por un valor** `p` **que no sea primo? No. Por ejemplo,** `n=40`**. Cuando** `p` **sea 4, ¿podrá ser** `n` **divisible por 4, es decir** `n%4==0`**? Después de dividir todas las veces posibles por 2, me queda** `n=5` **que ya no es divisible por 2, ni por tanto, por ningún múltiplo de 2. En general, al evaluar** `n%p`**,** `n` **ya ha sido dividido por todos los múltiplos de** `p`**.**

***Nota*. Podemos sustituir** `p++` **por** `p=p+2` **(tratando el primer caso** `p=2` **de forma aislada)**

El problema de factorizar un número (expresarlo como producto de factores primos) es importante.

Hasta la fecha, no se ha conseguido construir un algoritmo *que termine en un tiempo razonable* para factorizar un número cuando éste es grande[1] (ni se espera que se consiga)

¿Dónde se aplica? En *Criptografía*, para ocultar nuestros datos en Internet (por ejemplo, el número de la tarjeta de crédito). Mientras no se diseñe un algoritmo en este sentido, nuestros datos bancarios están bien protegidos de los *sniffers*.

Consultad:

```
http://angelrey.wordpress.com/2009/05/27/
       numeros-primos-criptologia-y-codificacion/
Google, Wikipedia -> Integer factorization, RSA numbers
```

[1] **Los conceptos de *número grande* y *tiempo razonable* se presentaron en la página 93**

2.2.5. OTRAS (PERNICIOSAS) ESTRUCTURAS DE CONTROL

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`
- ▷ `exit`

Durante los 60, quedó 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.

En esta asignatura, no se permitirá el uso de ninguna de las sentencias anteriores, excepto la sentencia `break` con el propósito aquí descrito dentro de la sentencia `switch`.
En caso contrario, el Suspenso está garantizado.

Referencias:

Teorema de Bohm y Jacopini -1966-. Flow diagrams, Turing machines and languages only with two formation rules. Communications of the ACM, 1966. Vol. 9, No.5, pp. 366-371

Dijkstra, E.W. Goto statement considered harmful. Communications of the ACM, 1968. Vol. 11, No.3, pp. 147-148

Para buenas normas de programación, consultad:

`http://www.geocities.com/aplicacionesjava/`

***Nota*. El anterior enlace incluye una referencia a un informe en el que se reconoce que un uso inadecuado de un `break`, provocó un apagón telefónico de varias horas en los 90 en NY.**