Maestría en Tecnología de Cómputo

Curso propedéutico de Programación

Tema I: Entrada salida, instrucciones de control y ciclos.

Conferencia No. 2

Título: Manejo de entrada y salida simple. Instrucciones de control y ciclos.

Objetivo: Identificar y utilizar las instrucciones de entrada y salida, así como las estructuras de control alternativas y repetitivas en el lenguaje C# para la implementación de algoritmos de secuencia no lineal.

Sumario:

Introducción

Con frecuencia los programadores se enfrentan a problemas cuyas soluciones no pueden resolverse con simples algoritmos secuenciales, teniendo que recurrir a estructuras más complejas como las llamadas estructuras de control.

Las estructuras de control pueden agruparse en tres categorías: las sentencias alternativas, las repetitivas y las de salto.

Las sentencias **alternativas** permiten **elegir uno de entre varios caminos en función de ciertas condiciones**. En algunos casos esta selección viene determinada por la evaluación de una expresión lógica. Este tipo de sentencias se dividen en tres:

- **if**
- **if-else**
- **switch**

1. Sentencias alternativas

Las *estructuras de alternativas* permiten verificar una o más condiciones y, en dependencia de su valor veritativo o lógico, ejecutar unos u otros pasos.

1.1. Alternativa simple

La alternativa simple verifica el valor veritativo de una condición determinada y, en dependencia de dicho valor, ejecutará o no un grupo de instrucciones. Se representa de la siguiente manera:

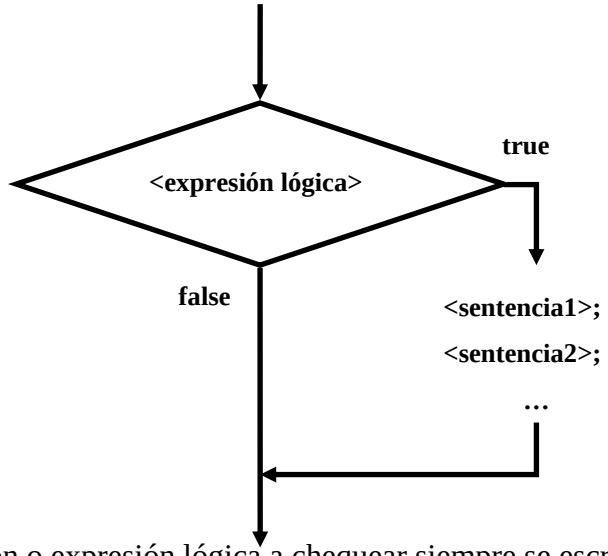
Sintaxis:

if (<expresión lógica>)

<sentencia>;

Esta estructura permite evaluar una condición o expresión lógica y si el resultado es verdadero, o sea, se satisface la condición, se ejecuta la sentencia que le sigue. De otra forma, si la expresión lógica es falsa no se ejecuta la sentencia y se salta directamente a la próxima sentencia que aparezca. La sentencia puede ser única o un bloque de sentencias como se muestra a continuación:

```
if ( <expresión lógica> )
{
        sentencia1;
        sentencia2;
        …
}
```



Note que la condición o expresión lógica a chequear siempre se escribe entre paréntesis.

Ejemplo

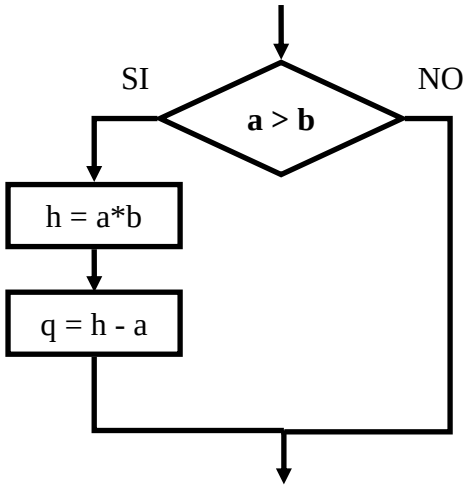
```
if (a > b)
{
   h = a * z;
   q = h – a;
}
```

También puede escribirse como sigue:

```
if (a > b) {  h = a * z;  q = h – a; }
```

pero evidentemente la primera forma resulta mucho más clara.

En este ejemplo se garantiza que sólo se calcule *h* como el producto de *a* por *z* y *q* como la diferencia de *h* y *a* si *a* fuera mayor que *b*. En caso contrario estos cálculos no se hacen.



Nótese que la alternativa altera el orden de la ejecución, ya que no se ejecutan las tres instrucciones de asignación, sino que, en dependencia del valor veritativo (lógico) de la condición (expresión), se ejecutan unas u otras.

Siguiendo el ejemplo visto en la Conferencia 1 sobre el cálculo del volumen de un cilindro hueco, ¿qué se mostraría al usuario si el programa se ejecuta con los juegos de datos siguientes?.

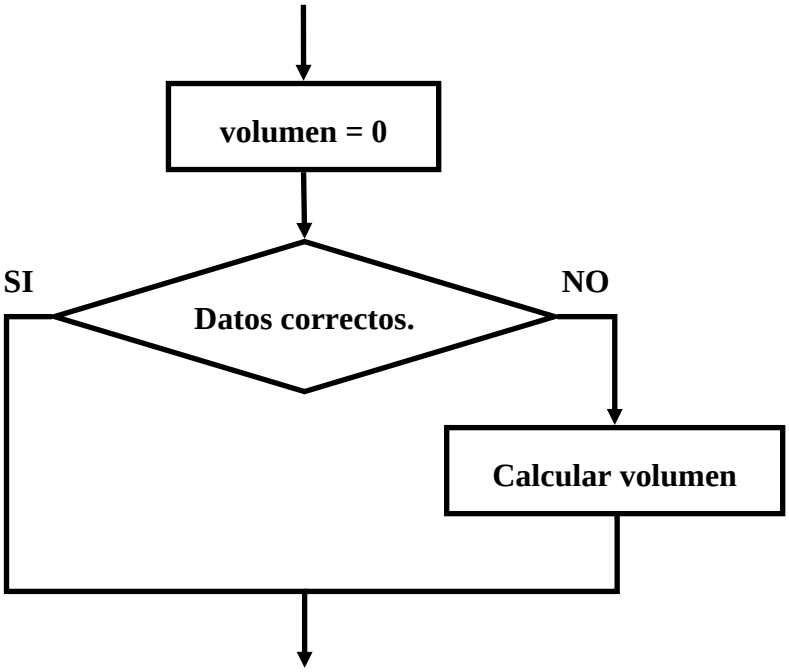
a) RadioExterior= 10 cm RadioInterior = 5 cm Longitud = 2 cm
b) RadioExterior = 2 cm RadioInterior = 8 cm Longitud = 4 cm
c) RadioExterior = 0 cm RadioInterior =3 cm Longitud =7 cm
d) RadioExterior = -10 cm RadioInterior = -5 cm Longitud = 3 cm
e) RadioExterior = 3 cm RadioInterior =1 cm Longitud = 0 cm
f) RadioExterior = 3 cm RadioInterior =0 cm Longitud = 10 cm

Respuestas:

a) 471 cm^3, un resultado correcto
b) -753,6 cm^3 un volumen no puede ser un valor negativo. Sucede porque el Radio Exterior es menor que el Radio Interior
c) -197,82 un volumen no puede ser un valor negativo. Sucede porque el Radio Exterior es menor que el Radio Interior. Además el Radio Exterior es cero lo cual no es posible, máxime siendo el radio Interior mayor que cero.
d) 471 cm^3, un resultado aparentemente correcto pero en realidad los datos son erróneos, pues los valores de los radios no pueden ser menores que ceros
e) 0 cm^3, un resultado correcto, pues si el cilindro es de longitud cero, evidentemente su volumen es cero.
f) 282,6 cm3, un resultado correcto, pues si el cilindro interior es de radio cero, evidentemente se trata de un cilindro no hueco.

Los errores que podrían cometerse al establecer valores incorrectos pueden evitarse si se incluye todo el cálculo dentro de una estructura condicional para validar que el Radio Exterior sea mayor o igual que el Radio Interior y que ambos y la longitud sean mayores o iguales que cero. En el caso extremo en que los radios o la longitud sean ceros el volumen tendrá un valor correcto, o sea, cero.

Para solucionar este problema será necesario modificar el algoritmo, de manera que se analicen primero los datos de entrada y sólo en el caso en que estén correctos se proceda al cálculo del volumen. Nótese que se trata entonces de un algoritmo no lineal como se muestra a continuación:

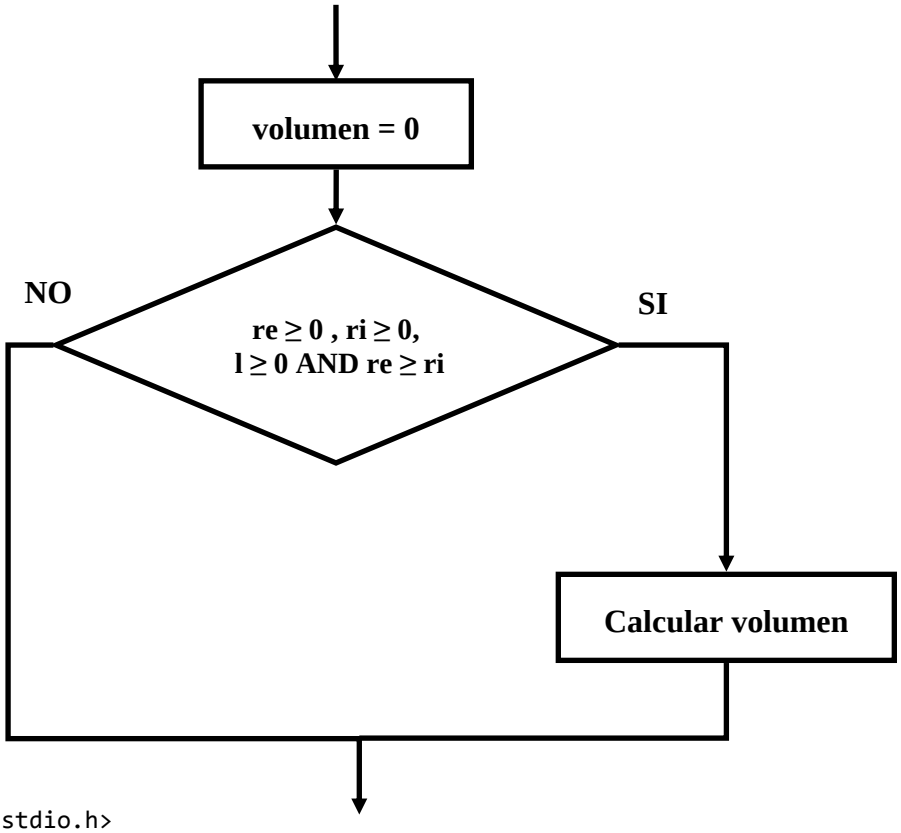


Aun cuando los datos no son correctos el valor más indicado para el volumen es cero, que puede interpretarse como que no existe un cilindro por lo que tampoco puede haber un volumen asociado.

Los datos son correctos si:

- radio exterior >= 0 y radio interior >= 0 y longitud >= 0 y
- radio exterior >= radio interior

Por tanto chequear en el algoritmo que los datos sean correctos implica chequear que se satisfagan estas condiciones, o sea, que sean verdaderas. Pero, en qué orden lógico deben ser chequeadas. Como es evidente, primero debe chequearse si los tres datos son mayores o iguales a cero y sólo si esta condición se satisface debe chequearse si el radio exterior es mayor o igual al interior. Observe que el radio exterior puede ser mayor o igual que el interior y sin embargo ambos o uno de ellos o incluso la longitud ser negativos, en cuyo caso estamos en presencia de datos de entrada erróneos, por lo que invertir el orden en que se chequean las condiciones puede representar una pérdida de tiempo y esfuerzo. Teniendo en cuenta estas consideraciones el algoritmo anterior puede modificarse para especificar los pasos con más detalles como se muestra a continuación:

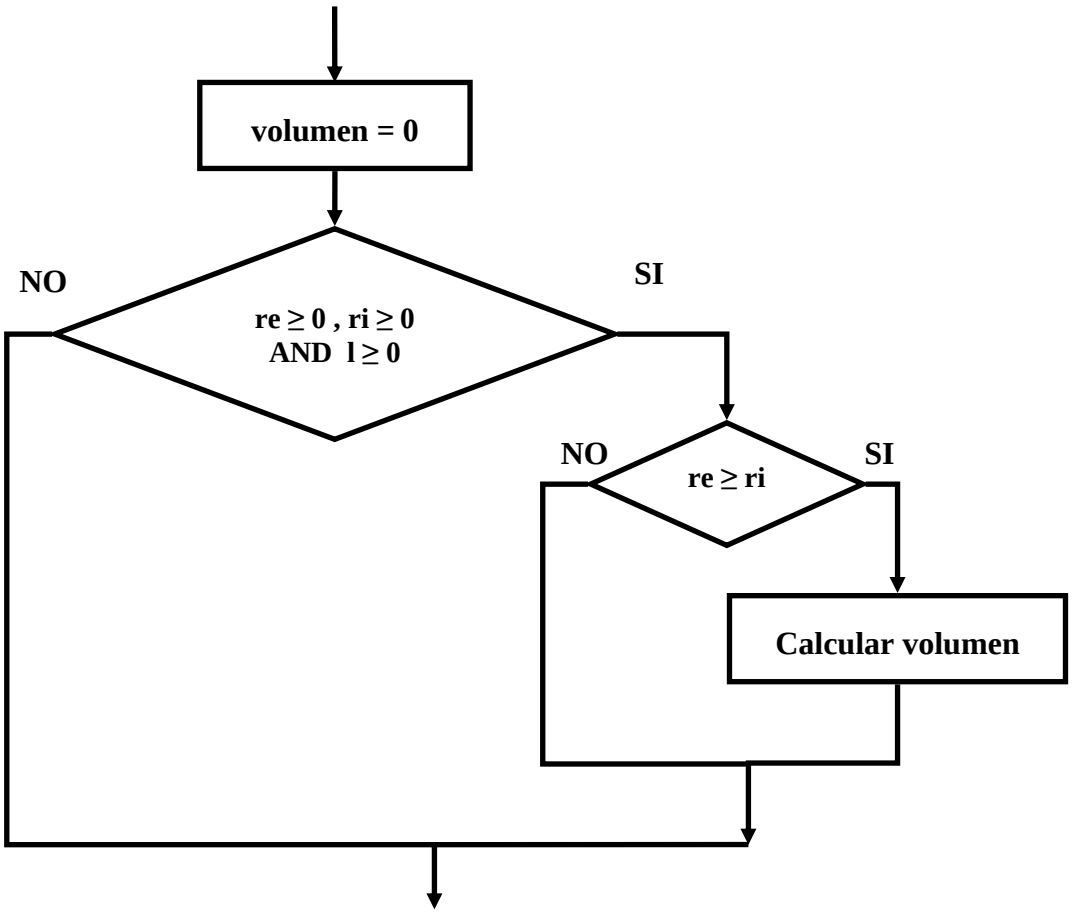



```
#include <stdio.h>
#define PI 3.1416f

float AreaBaseCilindro(float);
float VolumenCilindroHueco(float, float, float);

main()
{
   float radioExterior = 0;
   float radioInterior = 0;
   float longitud = 0;
   printf("Entre el radio exterior: \n");
   scanf("%g", &radioExterior);
   printf ("Entre el radio interior: \n");
   scanf("%g", &radioInterior);
   printf ("Entre la longitud: \n");
   scanf("%g", &longitud);
   float volumen = 0;
   if (radioExterior >= 0 && radioInterior>= 0 && longitud >= 0 &&
    radioExterior >= radioInterior)
   {
      float areaBaseExterior = AreaBaseCilindro(radioExterior);
      float areaBaseInterior = AreaBaseCilindro(radioInterior);
      float areaBase = areaBaseExterior - areaBaseInterior;
      volumen = areaBase * longitud;
   }
   printf("El volumen del cilindro hueco es: %g\n", volumen);
}
```

O de esta manera:




```
main()
{
   float radioExterior = 0;
   float radioInterior = 0;
   float longitud = 0;
   printf("Entre el radio exterior: \n");
   scanf("%g", &radioExterior);
   printf ("Entre el radio interior: \n");
   scanf("%g", &radioInterior);
   printf ("Entre la longitud: \n");
   scanf("%g", &longitud);
   float volumen = 0;
   if (radioExterior >= 0 && radioInterior>= 0 && longitud >= 0 )
      if (radioExterior >= radioInterior)
      {
         float areaBaseExterior = AreaBaseCilindro(radioExterior);
         float areaBaseInterior = AreaBaseCilindro(radioInterior);
         float areaBase = areaBaseExterior - areaBaseInterior;
         volumen = areaBase * longitud;
      }
   printf("El volumen del cilindro hueco es: %g\n", volumen);
}
```

1.2. Alternativas dobles

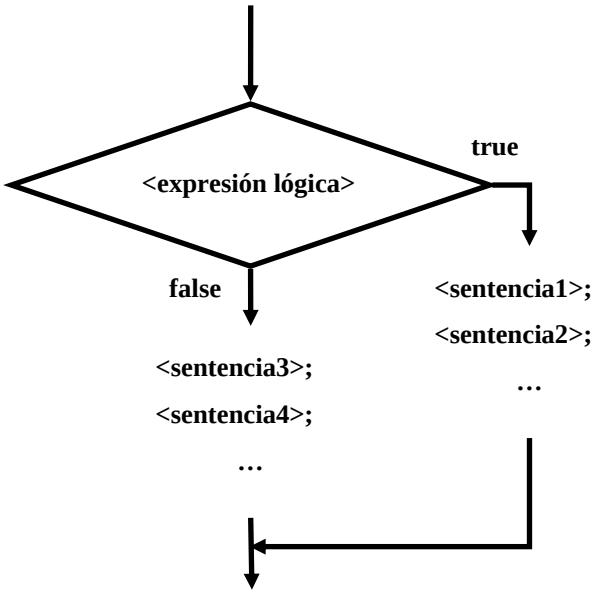
Se usa para especificar un conjunto de pasos a ejecutar si se cumple la condición que se chequea y otro conjunto diferente de pasos si ésta no se cumple, lo cual es una situación que se presenta muy frecuentemente.

Sintaxis:

if (<expresión lógica>**)**
    <sentencia1>;
else
    <sentencia2>;

Se lee, si la expresión lógica es verdadera se ejecuta la sentencia 1 en caso contrario se ejecuta la sentencia 2.

Ambas sentencias pueden ser o no bloques de sentencias, como muestra en el siguiente diagrama.

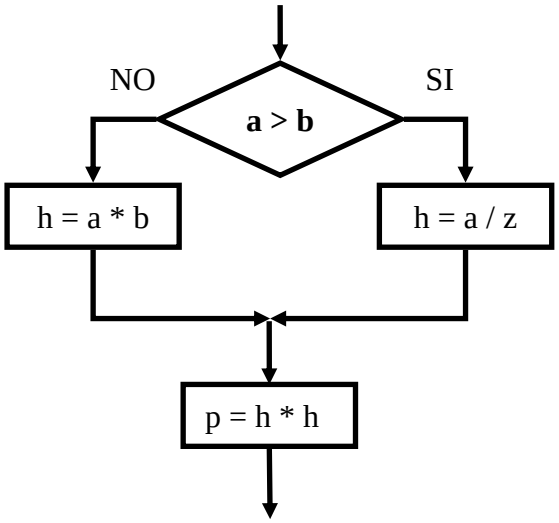


if (<expresión lógica>**)**
{
    <sentencia1>;
    <sentencia2>;
    …
}
else
{
    <sentencia3>;
    <sentencia4>;
    …
}

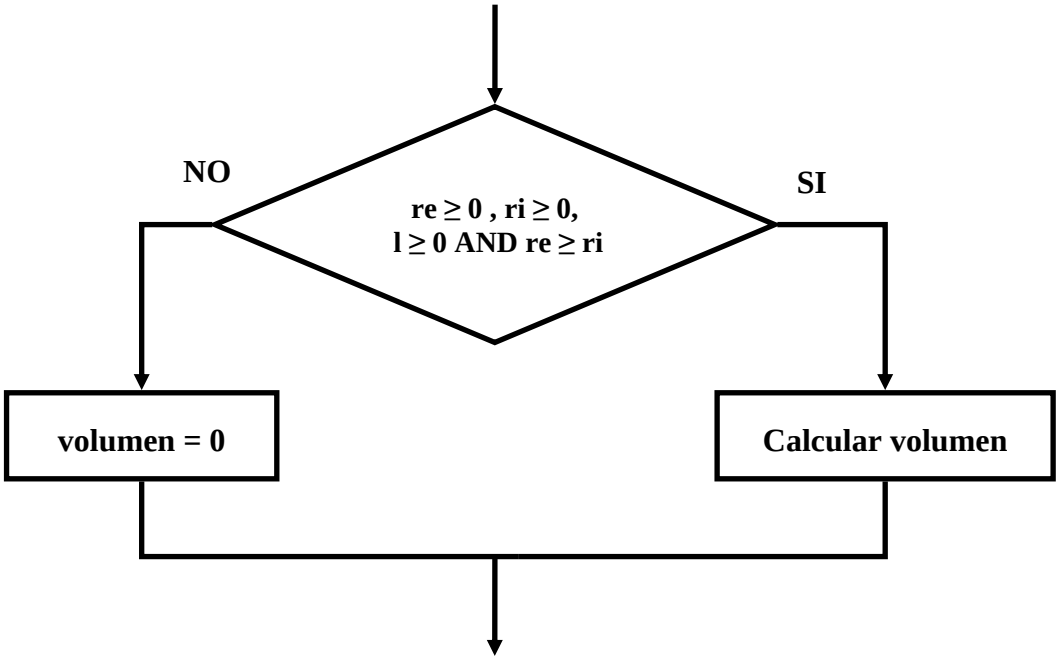
Ejemplo:

```
float h;
if (a > b) /*observe que esta sentencia requiere la declaración
previa de la variable h*/
   h = a * z;
else
   h = a / z;    //en ambos casos las sentencias son simples.
float p = h * h;
```

En este ejemplo según la relación entre *a* y *b* se calcula *h* de una forma u otra. Si *a* es mayor que *b*, o sea, si se satisface la condición *a* > *b*, se calcula *h* como el producto de *a* y *z* y a continuación se calcula *p* como *h* al cuadrado. Sin embargo, si *a* es menor o igual que *b*, o sea, si no se satisface la condición *a* > *b*, se calcula *h* por el cociente entre *a* y *z* y a continuación se calcula *p* como el cuadrado de *h*. Nótese que *h* se calcula una sola vez ya que o *a* es mayor que *b* o es menor o igual a *b*, pero nunca ambas cosas. Por otra parte, el cálculo de *p* es una sentencia que está fuera del **if** y a continuación de éste, por tanto, independiente de la forma en que se calcule *h* siempre se calculará *p* a continuación como el cuadrado de *h*.



El algoritmo para el cálculo del volumen puede modificarse utilizando una alternativa doble, la cual se muestra a continuación:




```
main()
{
   float radioExterior = 0;
   float radioInterior = 0;
   float longitud = 0;
   printf("Entre el radio exterior: \n");
   scanf("%g", &radioExterior);
   printf ("Entre el radio interior: \n");
   scanf("%g", &radioInterior);
   printf ("Entre la longitud: \n");
   scanf("%g", &longitud);
   float volumen ;
   if (radioExterior >= 0 && radioInterior>= 0 && longitud >= 0 && (radioExterior >=
radioInterior))
   {
      float areaBaseExterior = AreaBaseCilindro(radioExterior);
      float areaBaseInterior = AreaBaseCilindro(radioInterior);
      float areaBase = areaBaseExterior - areaBaseInterior;
      volumen = areaBase * longitud;
   }
   else
      volumen = 0;
   printf("El volumen del cilindro hueco es: %g\n", volumen);
}
```

Tanto para la alternativa simple como para la doble, es posible tener otra alternativa dentro de los pasos a realizar por cualquiera de los dos caminos, o sea, que la sentencia a ejecutar sea otro **if**. A esto se le llama estructuras alternativas anidadas. Para lograr mayor claridad en la lectura se recomienda no anidar por el camino que debe ejecutarse cuando se satisface la condición.

1.3. Operador condicional

Es el único operador incluido en C que toma 3 operandos, y se usa así:

<condición> ? <expresión1> : <expresión2>

El significado del operando es el siguiente: se evalúa **<condición>**, si es cierta se devuelve el resultado de evaluar **<expresión1>**, y si es falsa se devuelve el resultado de evaluar **<expresión2>**. Un ejemplo de su uso es:

```
b = (a > 0) ? a : 0;  // Suponemos a y b de tipos enteros
```

En este ejemplo, si el valor de la variable a es superior a 0 se asignará a **b** el valor de **a**, mientras que en caso contrario el valor que se le asignará será 0.

Hay que tener en cuenta que este operador es asociativo por la derecha, por lo que una expresión como **a ? b : c ? d : e** es equivalente a **a ? b : (c ? d : e).**

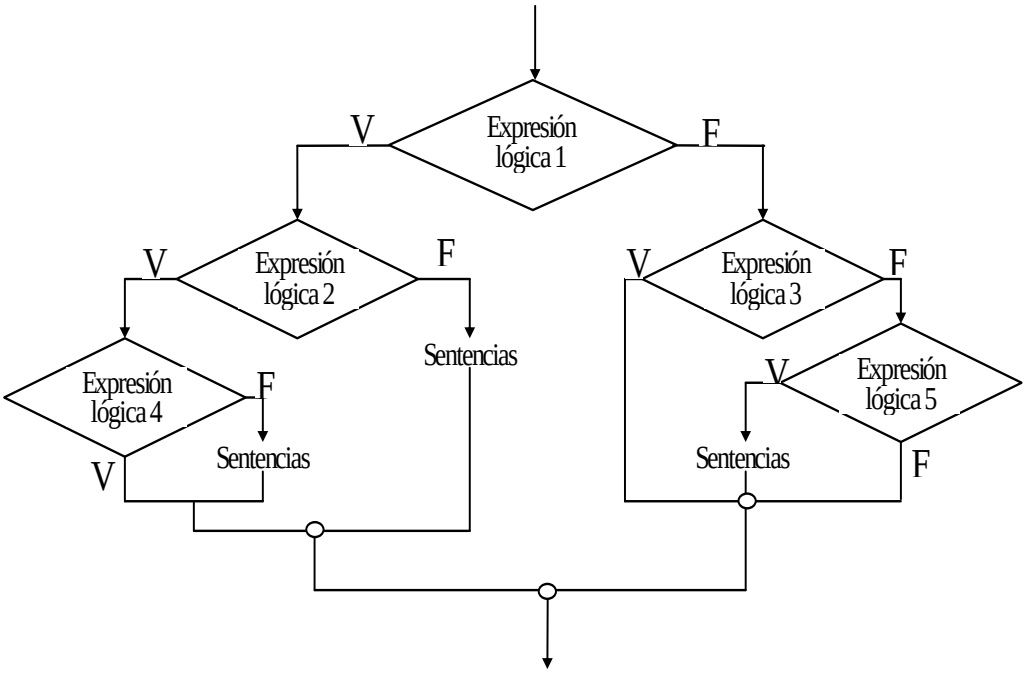
1.4. Alternativa múltiple

Con mucha frecuencia la selección de una alternativa no es tan simple. A veces se requiere hacer sucesivos chequeos de distintas condiciones antes de realizar una operación concreta. Otras veces encontramos distintos casos en que se puede presentar un problema de manera que cada uno demanda acciones diferentes. Tal es el caso, por ejemplo, de la conducta a seguir ante un semáforo, que depende de la luz proyectada en un instante concreto. A este tipo de situaciones, en programación, se les conoce como alternativa múltiple.

En C se utilizan las estructuras **if-else** anidadas o la sentencia **switch** para programar algoritmos con tales requerimientos.

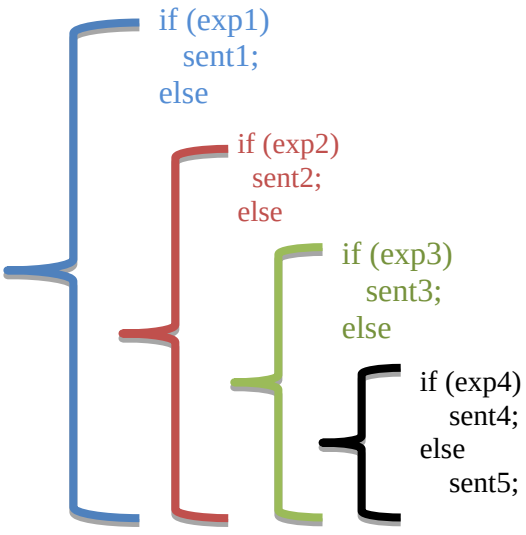
1.4.1. Sentencias if-else anidadas.

Gráficamente una situación como esta podría representarse de la siguiente manera:



Como el gráfico lo indica la sentencia a ejecutar por uno de los caminos de una alternativa, es otra alternativa y se puede programar anidando estructuras alternativas. Esto se puede expresar en un algoritmo de diferentes maneras.

Variante A.



Próxima sentencia

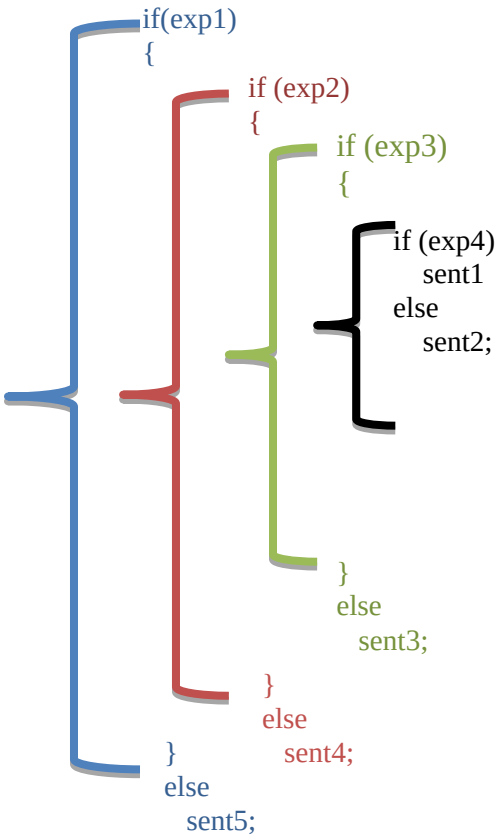
Se han usado colores y llaves para hacer más explícita la estructura en cuanto al casamiento **if-else**.

En este caso si la expresión lógica 1 es verdadera se ejecuta la sentencia 1, pero si es falsa se verifica la expresión lógica 2. Si expresión lógica 2 resulta verdadera se ejecuta la sentencia 2, pero si es falsa se chequea la expresión lógica 3 y así sucesivamente hasta alcanzar el final de la sentencia anidada.

Debe resaltarse que:

- Si se ejecuta la sentencia 5 es porque ninguna de las expresiones fue verdadera.
- Si se ejecuta alguna de las sentencias se pasa automáticamente a ejecutar *próxima sentencia*, pues eso significa que alguna expresión lógica resultó verdadera y por tanto ya no se necesita seguir chequeando otras condiciones.
- Al chequear la correspondencia **if-else**, debe razonarse de la misma manera que al chequear paréntesis dentro de una expresión. El casamiento se hace de adentro hacia fuera.

Variante B.



Próxima sentencia

La diferencia de la variante B con la A es que el anidamiento se ha hecho por las ramas verdaderas. El resultado es un algoritmo menos claro que el anterior, pues una misma construcción **if-else** queda dispersa dentro del algoritmo por lo que se necesita un mayor esfuerzo para entender el código, lo cual implica una pérdida de claridad en el algoritmo. Por tanto, se recomienda tratar, siempre que sea posible de programar con el estilo de la Variante A, para lo cual sólo se necesita en la mayoría de los casos escribir las expresiones lógicas a la inversa. Recuérdese que en todos los casos, cualquiera de las sentencias puede ser un bloque de sentencias, en cuyo caso se encerrarían entre {}.

La variante B gana en claridad, si la sentencia a ejecutar por la rama verdadera aun cuando no sea compuesta se encierra entre {}. Por ejemplo como se muestra a continuación:

Ejemplos

1.
```
if (a < b) //si a >= b no se ejecuta nada
    if (c > d)
    z = 5;
else
    z = 7;
```

A esto se le conoce como el problema del *else* colgante. El problema radica en que no se ve muy claramente a quién corresponde el *else.*

2.
```
if (a < b && c > d) // si a >= b se ejecuta z = 7
      z = 5;
  else
      z = 7;
```

El segundo caso **NO** es equivalente al código anterior.

Seudocódigo para la sentencia if-else

SI <condición> **ENTONCES**

instrucciones

SI NO

otras instrucciones

FIN SI

1.4.2. Sentencia Switch

También se presentan alternativas múltiples cuando tenemos que decidir qué camino tomar en la solución de un problema teniendo en cuenta que se está frente a uno de varios casos posibles. Como se vio anteriormente esto se puede programar con varios **if-else** anidados.

Pero cuando existen varios niveles de anidamiento se compromete la claridad del código. Para esta alternativa múltiple C# cuenta con la estructura **switch**. Su sintaxis es la siguiente:

switch (<selector> **)**

{

case <expresión constante> **:** <sentencia>**; break;**

.

.

default : <sentencia>**; break;**

}

donde:

expresión constante: es un posible valor del selector. Para cada valor listado se debe colocar la palabra reservada **case** y deben separase por **:**

Cuando se usa la sentencia **switch** el control se transfiere al punto etiquetado con el **case** cuya expresión constante coincida con el valor de la variable del **switch**. A partir de ese punto todas las sentencias serán ejecutadas hasta el final del **switch**, es decir hasta llegar a la llave cerrada **}**. Esto es así porque las etiquetas sólo marcan los puntos de entrada después de una ruptura de la secuencia de ejecución, pero no marcan las salidas.

Esta característica nos permite ejecutar las mismas sentencias para varias etiquetas distintas, y se puede eludir usando la sentencia de ruptura **break** al final de las sentencias incluidas en cada **case**.

Si no se satisface ningún **case**, el control pasará a la sentencia después de la etiqueta **default**. Esta etiqueta es opcional y si no aparece se abandonará el **switch**.

Ejemplos:

```
#include <stdio.h>

main()
{
   int op;
   printf("Entre un numero del 1 al 3\n");
   scanf("%d", &op);
   switch (op)
   {
      case 1: printf("Opcion escogida: %d\n", op);
      break;
      case 2: printf("Opcion escogida: %d\n", op);
      break;
      case 3: printf("Opcion escogida: %d\n", op);
      break;
      default: printf("Opcion no valida");
      break;
   }
}
```

Seudocódigo para la sentencia Switch

SEGÚN SEA <variable>

CASO = <condición>

CASO = <condición>

instrucción o instrucciones

OTRO CASO

instrucción o instrucciones

FIN SEGÚN

2. Sentencias repetitivas

Las sentencias repetitivas permiten ejecutar varias veces una instrucción o un bloque de instrucciones mientras se cumpla una condición. Es decir, permiten definir bucles donde ciertas instrucciones se ejecuten varias veces. A continuación se describen cuáles son las instrucciones de este tipo incluidas en C.

2.1. Tipos de ciclos

Existen distintos tipos de ciclo, los cuales serán utilizados en dependencia de la situación concreta que se desea resolver. Todos verifican, de alguna manera, el cumplimiento de una condición para determinar si el ciclo debe detenerse o no.

En algunos algoritmos se conoce la cantidad de veces que un conjunto de pasos deben repetirse, en otros se conoce una condición de parada. Teniendo en cuenta este criterio, los ciclos pueden clasificarse en:

- Ciclos por condición: y estos a su vez de acuerdo al momento en que se verifica el cumplimiento de la condición, con respecto a la ejecución del conjunto de pasos, puede subdividirse en:
    - Ciclo con precondición
    - Ciclo con poscondición
- Ciclos por variable de control

2.1.1. Ciclos por condición

Ciclo con precondición

Sintaxis:

while ([<expresión lógica>])
  <sentencia>;

o si tiene más de una sentencia:

while ([<expresión lógica>])
{
  <sentencia1>;
  <sentencia2>;
  …
}

Lo primero que se hace es evaluar la expresión lógica, si es verdadera se ejecutan todas las sentencias que aparecen dentro de la estructura repetitiva y se vuelve a evaluar la expresión lógica manteniendo el mismo comportamiento. Si la expresión lógica es falsa no se ejecutan las sentencias que aparecen dentro de la estructura repetitiva, sino que se pasa a la primera sentencia que aparezca después de ella. Si la condición siempre es verdadera (por ejemplo: while(true)) , el bucle se ejecutará indefinidamente. Si la primera vez que se evalúa la condición resulta falsa, la sentencia no se ejecutará ninguna vez.

Ciclo con poscondición

Sintaxis:

do
    <sentencia>;
while ([<expresión lógica>]);

o si tiene más de una sentencia:

do
{
  <sentencia1>;
  <sentencia2>;
  …
}**while** ([<expresión lógica>]) ;

Aquí, por el contrario, lo primero que se hace es ejecutar las sentencias que aparecen entre las palabras reservadas **do** y **while** y a continuación se evalúa una expresión lógica que, de ser falsa hace que se abandone el ciclo, y si es verdadera hará que se transfiera el control a la primera sentencia después de la palabra reservada **do**. Si la condición siempre es verdadera(por ejemplo: while(true)) , el bucle se ejecutará indefinidamente. A diferencia del bucle **while**, la evaluación se realiza después de ejecutar la sentencia, de modo que siempre se ejecutará al menos una vez.

En ambos casos es necesario que:

1. al momento de evaluar la expresión lógica, todas las variables involucradas hayan tomado valor.
2. siempre se alcance, en algún momento, la condición de parada de lo contrario resultaría un ciclo infinito.

Ejemplo:

El programa que se muestra a continuación recibe un entero e incrementa su valor mientras sea menor que 100.

Varinte con **while**

```
main()
{
 int x;
 printf("Entre un numero entero\n");
 scanf("%d", &x);
 /*incrementa el valor de x mientras x sea menor que 100.*/
 while (x < 100)
    x++;
    printf("Luego del ciclo, x es igual a %d\n", x);
}
```

Varinte con **do – while**

```
main()
{
 int x;
 printf("Entre un numero entero\n");
 scanf("%d", &x);
 /*incrementa el valor de x hasta que x sea menor que 100.*/
 do
    x++;
 while (x < 100);
    printf("Luego del ciclo, x es igual a %d\n", x);
}
```

Seudocódigo para los ciclos por condición.

Instrucción en C	Seudocódigo
while([<expresión lógica>]) <sentencias>;	**MIENTRAS** <expresión lógica> <sentencias> **FIN MIENTRAS**
do <sentencias>; **while**([<expresión lógica>])	**HACER** <sentencias> **MIENTRAS** <expresión lógica>

2.1.2. Ciclos por variable de control

Se utilizan cuando se conoce la cantidad de veces que se desea repetir un fragmento de código. Se usa una variable de control que se inicializa en un primer valor incrementándose/decrementándose en cada iteración hasta llegar al valor final. Esta variable de control frecuentemente se usa dentro de los pasos que se repiten, pero no se puede declarar ni alterar su valor dentro de la estructura repetitiva.

Sintaxis:

for ([<inicialización>]; [<condición>] ; [<incremento>])

<sentencia>;

La sentencia es ejecutada repetidamente hasta que la evaluación de la condición resulte falsa. Se comienza desde los valores indicados en la inicialización y en cada nueva iteración se incrementa según lo que se especifique en el incremento. En lugar de una sola sentencia puede ser necesario repetir un bloque sentencias.

Antes de la primera iteración se ejecutará la inicialización del bucle. En este apartado se iniciarán las variables de control del bucle. Estas variables pueden ser declaradas en este punto, en cuyo caso su alcance es local al bucle "**for**". Después de cada iteración se ejecutará el incremento de las variables del bucle.

Todas las expresiones son opcionales, si no se especifica la condición se asume que es verdadera. Ejemplos:

```
for (int i = 0; i < 100; i++); //la sentencia se ejecutará 100 veces
```

Como las expresiones son opcionales, podemos simular bucles "**while**":

```
int i = 0;
for (; i < 100; )
i++;
```

o escribir bucles infinitos: `for (; ; ) ;`

Veamos en el siguiente ejemplo como imprimir todos los números del 1 al 100 utilizando un ciclo **for:**

```
main()
{
 for(int x = 1;x <= 100; x++)
  printf("%d \n", x);
}
```

Seudocódigo para los ciclos por variable de control

Instrucción en C
for (<inicialización>; <condición>; <incremento>)   <sentencia>;
Seudocódigo
DESDE <inicialización> **HASTA** <condición> **INCREMENTO** <valor>   <sentencias> **FIN DESDE**

3. Algoritmos con ciclos

Cómo ya se ha dicho, en la práctica vemos con frecuencia muchos algoritmos con ciclos. Veamos algunos ejemplos.

a) Encontrar el menor número entero positivo mayor que *X* que sea múltiplo del entero positivo *Y*, siendo *X* mayor o igual que *Y*.

   Por ejemplo, si *X* toma valor 5 y *Y* toma valor 4, el primer número mayor que 5 que es múltiplo de 4 es 8.

   Esto es lo mismo que encontrar el menor número entero mayor que *X* que sea divisible por *Y*. Para resolver esto necesitamos incrementar X de 1 en 1 y para cada nuevo valor de *X* comprobar si la expresión *X* % *Y* = = 0 es verdadera en cuyo caso se dice que el nuevo valor *X* es el número buscado. En otras palabras, debemos incrementar *X* mientras no se satisfaga la condición *X* % *Y* = = 0.

   Veamos a continuación dos variantes de este algoritmo:

Variante 1

```
INICIO
   LEER X ,Y
   SI (X <= 0 OR Y <= 0 OR X < Y)
    ENTONCES
       No cumple
    SI NO
        X ← X + 1
      MIENTRAS ( X MOD Y <> 0 )
              X←X + 1
      FIN MIENTRAS
      IMPRIMIR X
    FIN SI
FIN
```

Variante 2

```
INICIO
   LEER X ,Y
   SI (X  <= 0 OR Y <= 0 OR X < Y)
   ENTONCES
       No cumple
   SI NO
     HACER
        X←X +1
     MIENTRAS ( X MOD Y  <>  0 )
       IMPRIMIR X
   FIN SI
FIN
```

Observar que:

- Las dos variantes resuelven el mismo problema de forma satisfactoria. En la primera el incremento se realiza sólo si se cumple la condición que se chequea al inicio, por lo que pudiera suceder que nunca se llegue a incrementar la variable pues el primer valor analizado sea un múltiplo de Y, mientras que en la segunda variante se realiza el incremento y luego, al final, se chequea la condición. En este caso el incremento siempre se realiza al menos una vez. Los programadores escogen una u otra forma de implementar cada ciclo según sea más conveniente o más claro o a veces simplemente como se les ocurra primero.
- Un algoritmo con ciclo debe garantizar que el ciclo pare en algún momento, pues de lo contrario sería un ciclo infinito que no permitiría continuar la ejecución del programa. En estos casos la variable *X* implicada en la condición va cambiando su valor dentro del ciclo, de manera que se garantiza que en algún momento se satisfaga la condición de parada. Es responsabilidad de los programadores velar por que los ciclos sean finitos.
- Las variables implicadas en la condición tomaron valor antes de que fuese necesario evaluar la expresión lógica por primera vez, lo cual garantiza que desde el principio ésta se evaluó con valores correctos dentro de la lógica del algoritmo.
- La variable X se inicializa con el número a partir del cual se desea buscar el múltiplo de Y y se va incrementando, por lo que se pierde el valor inicial. En este caso no es importante conservar el valor inicial, pues lo que interesa es el valor encontrado como múltiplo de Y.
- Sobre la variable *X* se ha implementado un proceso acumulativo, en este caso de suma. Nótese que este proceso tiene dos momentos importantes, primero la inicialización de la variable y dentro del ciclo, la suma o acumulación de que se trate. Recuerde que el compilador al ejecutar la asignación *X*= *X* + 1, primero evalúa la expresión de la derecha y el resultado lo asigna en la localización de memoria reservada para *X*, por lo que:
    - primero: busca el valor de *X*
    - segundo: a *X* le incrementa 1
    - tercero: asigna en *X* el resultado.

Por tanto, si a la variable *X* no se le asigna nada al inicio qué valor tendrá cuando se ejecute la expresión *X* + 1. En este caso el compilador accederá a esa localización de memoria y leerá como un entero lo que allí aparezca que puede ser cualquier valor. Algunos compiladores inicializan las variables en un valor por defecto según el tipo, por ejemplo, cero para los enteros, pero ni aun cuando esto suceda demos omitir la inicialización de las variables, pues hacemos el código más dependiente de una versión del compilador y por tanto menos portable.

- A este tipo de ciclo se le conoce como ciclo controlado por condición, pues en dependencia del cumplimiento de una condición se continúa o abandona el ciclo.

Código en C Variante 1.

```
#include <stdio.h>

main()
{
 int x, y;
 printf("Entre el primer valor \n");
 scanf("%d", &x);
 printf("Entre el segundo valor \n");
 scanf("%d", &y);
 if (x <= 0 || y <= 0 || x < y)
    printf("Ambos valores deben ser positivos, y el 1er valor debe ser mayor que el
segundo \n");
 else
 {
    x++;
    while (x % y != 0)
       x++;
    printf("El menor numero mayor que el 1er valor y multiplo del segundo es: %d
\n", x);
 }
}
```

Código en C Variante 2.

```
#include <stdio.h>

main()
{
 int x, y;
 printf("Entre el primer valor \n");
 scanf("%d", &x);
 printf("Entre el segundo valor \n");
 scanf("%d", &y);
 if (x <= 0 || y <= 0 || x < y)
    printf("Ambos valores deben ser positivos, y el 1er valor debe ser mayor que el
segundo \n");
 else
 {
    do
       x++;
```

```
    while (x % y != 0);
    printf("El menor numero mayor que el 1er valor y multiplo del segundo es: %d
\n", x);
 }
}
```

b) Calcular el factorial de un número entero positivo n.
El factorial de n se denota n! y se define como:
- para n= 0 factorial= 1
- para n > 0 factorial= producto de los n primeros números enteros. Ejemplo:

5! = 1 * 2 * 3 * 4 * 5 = 120

```
INICIO
SI n < 0  ENTONCES
        no está definido
      SI NO
      SI n = 0 ENTONCES
         factorial ←1
      SI NO
         factorial ←1
         DESDE i← 2 HASTA n
            factorial ← factorial * i
         FIN DESDE
            IMPRIMIR factorial
      FIN SI
FIN SI
FIN
```

Observar que:

- En este caso, a diferencia del anterior, se sabe de antemano cuántas veces se desea ejecutar el ciclo, o lo que es lo mismo se conocen los valores para los que se desea ejecutar el ciclo. En este ejemplo se necesita hacer el ciclo *n*-1 vez, o sea, desde 2 hasta *n* y para cada uno de los valores de *i* ejecutar el producto indicado.
- *i* toma todos los valores comprendidos entre 2 y *n*.
- en cada nueva iteración *i* se incrementa en 1. Por supuesto, igual podría tomar los valores desde *n* hasta 2 decrementándose en cada nueva iteración.
- cuando *i* toma valor *n*+1 se abandona el ciclo.
- La variable *i* funciona como variable de control del ciclo, y no debe cambiar su valor dentro de éste, pues de lo contrario no se podría controlar las veces que se hace el ciclo.
- Al igual que en el ejemplo anterior sobre la variable *factorial* ocurre un proceso acumulativo pero del producto en lugar de la suma. Por ese motivo se inicializa en 1, el elemento neutro del producto, de manera que no afecte el resultado final.
- Este algoritmo se puede hacer más eficiente si se obvia la condición *n* = = 0, nótese que tanto si esa condición se satisface o no, se le asigna 1 a la variable *factorial*. Por otra parte, si *n* es igual a cero nunca se ejecutaría el ciclo pues 2 es mayor que *n*, de la

misma manera sucede con *n* igual a 1, por tanto el algoritmo se puede simplificar inicializando la variable *factorial* en 1.

Código en C.

```
#include <stdio.h>

main()
{
  int n;
  printf("Entre el valor \n");
  scanf("%d", &n);
  if (n < 0)
     printf("No es posible calcular el factorial \n");
  else
  {
     int f = 1;
     for (int i = 2; i <= n; i++)
        f = f * i;
     printf("El factorial es: %d\n", f);
  }
}
```

Suponga ahora que se desea conocer los factoriales de los números de un intervalo dado. Para resolver esto es necesario programar un ciclo para calcular el factorial, como se hizo anteriormente, para cada número del intervalo. Nótese que la respuesta en este caso no es un solo valor sino varios de ellos posiblemente, por lo que será necesario que en la medida que los números son analizados debamos almacenar su factorial. Con las estructuras de datos conocidas hasta el momento esto no es posible, pues todas ellas permiten almacenar un valor simple y en este caso se trata de almacenar una colección de valores. En la próxima conferencia se comenzará el estudio de los arreglos en C, que son unas estructuras de datos que permiten dar solución al problema planteado.

4. Ciclos Anidados

Son ciclos que están dentro de otros ciclos, donde la ejecución de los ciclos internos depende de la ejecución de los ciclos externos. A continuación se muestran dos ciclo **for** anidados de 10 iteraciones cada uno, de manera que por cada iteración del ciclo externo ocurren 10 del ciclo interno. Note que las variables de control son distintas para cada ciclo.

```
for (int i = 0; i < 10; i++)
{
    for (int j = 0; j < 10; j++)
    {
   …
    }
}
```

También pueden anidarse ciclos de diferentes tipos:

```
do
{
      for(int i=0;i<10;i++)
```

```
      {
        ...
      }
}while(condicion);
```

En este caso mientras se cumpla la condición para el ciclo do/while se estará ejecutando el ciclo interno.

Veamos un ejemplo donde se muestra las tablas de multiplicar del 1 al 10:

```
#include <stdio.h>

main()
{
   for(int i = 1; i <= 10; i++)
      {
          printf("Tabla del %d\n", i);
          for(int j = 1; j <= 10; j++)
             printf("%d x %d = %d\n", i, j, (i * j));
      }
}
```

A continuación veremos un algoritmo que Simula el comportamiento de un reloj digital, imprimiendo la hora, minutos y segundos de un día desde las 0:00:00 horas hasta las 23:59:59 horas.

```
INICIO
        DESDE hora ← 0 HASTA 23
              DESDE minuto ← 0 HASTA 59
                    DESDE segundos ← 0 HASTA 59
                          Imprimir hora, minutos, segundos
                    FIN DESDE
              FIN DESDE
        FIN DESDE
FIN.
```

Código en C.

```
#include <stdio.h>

main()
{
   for(int i = 0; i <= 23; i++)
     for (int j = 0; j < 59; j++)
       for (int k = 0; k < 59; k++)
         printf("La hora es: %d:%d:%d\n", i, j, k);
}
```

5. Manejo de entrada y salida estándar

Como se ha ido viendo hasta el momento, la ejecución de un programa consta fundamentalmente de tres fases: entrada de datos, procesamiento de los datos, y salida de los datos.

Los datos de entrada estándar son datos numéricos o cadenas de caracteres. Los datos numéricos se almacenan en uno de los tipos de datos primitivos que se han estudiado. El almacenamiento de las cadenas de caracteres será estudiado en próximas clases.

La función printf, que se ha utilizado hasta ahora, permite escribir en pantalla, con un cierto formato. La especificación del formato comienza con el carácter especial %. Una especificación de formato está formada de la siguiente manera:

%[flags][ancho][.precisión][{h|l|L}]tipo

En la tabla se muestran los elementos de formato y su significado.

flags	**Significado**
-	Justifica a la izquierda, dentro del ancho especificado (por defecto se justifica a la derecha)
+	Antepone el signo + o – al valor de salida. Por defecto solo se pone el signo – a los valores negativos.
0	-rellena con 0 no significativos hasta alcanzar el ancho mínimo especificado
blanco	Antepone un espacio en blanco al valor de salida si este es positivo
#	Si se utiliza con la especificación o, x, o X, antepone al valor de salida 0, 0x, o 0X Si se utiliza con la especificación e, E o f, fuerza a que los valores tengan un punto decimal. Si se utiliza con la especificación g o G, fuerza a que los valores tengan un punto decimal, y evita que los ceros arrastrados sean truncados. Se ignora con c,d,i,u y s.
ancho	Mínimo número de posiciones para la salida. Si el valor no cabe en el ancho, este se incrementa
precisión	Depende de del tipo de la salida
tipo	Es uno de los caracteres que se muestran en la siguiente tabla.

tipo	**Significado**
d	Entero con signo en base 10
i	Entero con signo en base 10
u	Entero sin signo en base 10
o	Entero sin signo en base 8
x	Entero sin signo en base 16, con las letras en minúsculas
X	Entero sin signo en base 16, con las letras en Mayúsculas
f	Valor real de tipo double con signo de la forma [-]dddd.dddd (el número de dígitos decimales depende de la precisión, es 6 por defecto)
e	Valor real de tipo double con signo de la forma [-]d.dddde[+-]ddd
E	Valor real de tipo double con signo de la forma [-]d.ddddE[+-]ddd
g	Valor real de tipo double con signo, en formato f o g, el que se considere más

	compacto para el valor y precisión dados
G	Igual que g, pero el exponente e va en mayúsculas
c	Un único carácter, correspondiente al byte menos significativo
s	Escribe una cadena de caracteres hasta el 1er carácter nulo

El ancho y la precisión pueden ser sustituidos por el caracter *, en cuyo caso sus valores se toman de argumentos enteros. Por ejemplo:

int ancho = 15; int precision = 6; double valor = 3.1;

printf(“%*.*f”, ancho, precision, valor);

La precisión, como se mencionaba, depende del tipo de la salida. En la siguiente tabla se muestran los significados de la precisión, para cada tipo.

tipo	**Significado de la precisión**
d, i, u, o, x, X	Especifica el número mínimo de dígitos que se tienen que escribir. Rellena con 0 si es necesario. En caso de que el valor exceda a la precisión, no se trunca
e, E, f	Especifica el número de dígitos después del punto decimal. El valor se redondea a los dígitos deseados
g, G	Especifica el máximo número de dígitos significativos a escribir
c	No se aplica la precisión
s	Especifica el máximo número de caracteres a escribir. Si la cadena sobrepasa este número, los caracteres sobrantes son ignorados
h	Se utiliza como prefijo de los tipos d, i, o, x y X, para especificar que el argumento es short int, o con u si se desea especificar un short unsigned int
I	Se utiliza como prefijo de los tipos d, i, o, x y X, para especificar que el argumento es long int, o con u si se desea especificar un long unsigned int. También se utiliza con los tipos e, E, f, g y G para especificar un double antes que un float
L	Se utiliza con los tipos e, E, f, g y G para especificar un long double

No solo la salida tiene un formato en el lenguaje c, la entrada también lo tiene. El formato de la entrada interpreta los valores pasados como entrada. La función scanf, que hemos utilizado, interpreta los valores que se entran por teclado. Para ello, si se desea leer múltiples valores de una sola vez, es necesario especificar el formato de cada uno de ellos, separado por espacios en blanco.

Supongamos que se desean leer tres valores enteros, a, b y c. la función scanf debe ser escrita como: scanf("%d %d %d", &a, &b, &c)

Así, los valores entrados por teclado van a escribirse separados por espacios en blanco. Si quisieran escribirse separados por otro carácter (ej. La coma), la función scanf debe ser reescrita: scanf("%d , %d , %d", &a, &b, &c)

scanf es una función que tiene retorno. Dicho retorno es un entero, que se corresponde con la cantidad de datos leídos y asignados. En el siguiente ejemplo, se introducen 3 datos y se muestra en pantalla el total de datos leídos y asignados.

```
#include <stdio.h>

main()
{
   int a,b,c, r;
   r = scanf("%d , %d , %d", &a, &b, &c);
   printf("Total de datos leidos y asignados: %d \n", r);
}
```

La especificación de formato de la función scanf es como sigue:

%[*][ancho][{h/l}]tipo

La presencia del carácter * denota que el dato de entrada a que se hace referencia será leído pero no asignado (no se tomará en cuenta). El ancho es el máximo número de caracteres a leer de la entrada. El valor h se utiliza con los tipos d, i, n, o y x para especificar que el argumento es un short int, o con el tipo u para especificar que el argumento es un short unsigned int. De forma similar, el valor l se utiliza con los tipos d, i, n, o y x para especificar que el argumento es un long int, o con el tipo u para especificar que el argumento es un long unsigned int. Igualmente, con los tipos e, f y g, para especificar que el argumento es double. Los tipos a utilizar en la función scanf se detallan a continuación.

tipo	**argumento es puntero a**	**Significado**
d	int	Entero con signo en base 10
o	int	Entero con signo en base 8
X, X	int	Entero con signo en base 16
i	int	Enteros con signo en base 10, 8 o 16. Si empieza con 0 se asume base 8, si empieza con 0x o 0X se asume base 16
u	unsigned int	Entero sin signo en base 10
f,e, E, g, G	float	Valor con signo de la forma [-]d.dddd[{e|E}[+-]ddd
c		Un único carácter
s		Escribe una cadena de caracteres

Si quisiéramos leer solamente un carácter, también podemos utilizar la función getchat, proporcionada por C. Igualmente, es posible escribir un carácter en pantalla, con la función putchar.

Conclusiones

No todos los algoritmos constituyen secuencias lineales de pasos. En la mayoría de los casos los programadores tienen que enfrentar algoritmos con mayores niveles de complejidad en los que se producen ramificaciones según van ocurriendo las cosas, entre otras razones por la ocurrencia de condiciones extremas. En estos casos debemos programar nuestros algoritmos para que funcionen correctamente independientemente del juego de datos con que se corra.

La mayoría de los lenguajes brindan distintas posibilidades para programar algoritmos con alternativas, es responsabilidad del programador conocerlas y utilizar adecuadamente su sintaxis para lograr los resultados esperados. Dentro de estas estructuras encontramos if, if-else y switch.

Es muy frecuente para los programadores enfrentarse a algoritmos en los que es necesario repetir varias veces un segmento de pasos hasta que se cumplan determinadas condiciones. Según el contexto del problema se puede escoger alguno de los patrones estudiados:

- "hacer mientras se cumpla la condición"(do-while): Primero se hace lo que depende de la condición y luego se comprueba si esta se cumple. Este caso admite al menos una iteración aunque no se cumpla la condición.
- "mientras se cumpla la condición, hacer"(while): Primero comprueba la condición y luego hace lo que está dentro del ciclo hasta que esta se cumpla. Si desde el inicio no se cumple la condición no ocurriría ninguna iteración.
- "para tales condiciones iniciales hacer hasta que se cumplan tales condiciones finales"(for). Se utiliza una que se suele llamar variable de control. A este variable se le da un valor inicial. Luego el ciclo depende de una condición en la que está involucrada dicha variable. Dentro del ciclo la variable de control cambia de valor de modo tal que en algún momento se deje de cumplir la condición de repetición y el ciclo termina.

En esta conferencia hemos resuelto algunos problemas con relativa complejidad en su comprensión y análisis pero muy simples de algoritmizar. Es importante hacer hincapiés en la legibilidad del código para que otros programadores o nosotros mismos cuando queramos hacerle modificaciones o utilizar partes de él en otros programas, podamos entenderlo fácilmente.