Sentencias de control. Estructuras de repetición

Mediante las estructuras iterativas se puede hacer que un bloque de instrucciones se repita un número de veces.

La estructura iterativa.

Con las estructuras algorítmicas vistas hasta ahora no se puede expresar todas aquellas acciones que requieran una repetición de las mismas durante una cantidad de tiempo determinada. Es necesario conocer otra estructura que permita la repetición de las acciones que queramos.

Repetición con condición inicial.

Pensemos en lo que ocurre cuando, para venir a la Universidad, llegamos a la parada del autobús, lo lógico es que esperemos en ella hasta que llegue un autobús con plazas libres, más o menos algo así:

```
algoritmo Coger un autobús
    llegar a la parada
    mientras no haya autobús con plazas
         esperar a que llegue uno
    subir al autobús
falgoritmo.
```

Si se observa el algoritmo anterior se detecta que aparece la palabra **mientras**, la cual proporciona la idea de repetición *-si no hay autobuses, debemos esperar-* que es lo que se hace realmente cuando se llega a la parada y no hay ningún autobús al que subir.

La sintaxis que se corresponde con esta nueva estructura es la siguiente:

```
while (condición)
      S;
```

Esto es, la sentencia **S** se repetirá **mientras** la condición sea cierta. Si la condición es falsa ya desde un principio, la sentencia no se ejecuta nunca. Si queremos que se repita más de una sentencia, basta agruparlas entre **{** y **}**.

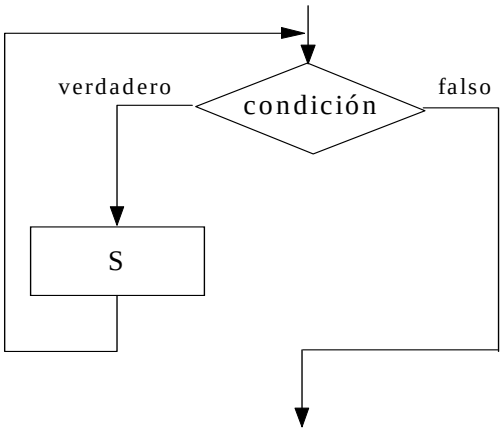
```
while (condición){
      S;
}
```

Donde **condición** representa una expresión booleana que:

- Si es **cierta** indica que el conjunto de instrucciones denotadas por *S* debe volver a ejecutarse de nuevo.
- Si es **falsa** indica que el flujo de control del algoritmo debe transferirse a la instrucción inmediatamente siguiente a la llave de cierre.

A este tipo de construcciones algorítmicas se las denomina **bucles**, a la condición de este tipo de bucles se le llama **condición de continuación**, y a las instrucciones que se ejecutan de manera repetida se las conoce como **cuerpo** del bucle.

El *dfd* asociado a esta sentencia de repetición lo podemos ver en la siguiente figura.



Tiene que haber dentro de las instrucciones del cuerpo del bucle algún elemento que altere el valor de la condición, pues de lo contrario se entraría en un bucle infinito.

✓ ¿Qué hace el siguiente fragmento de código?

```
int num;
num=0;

while (num != 99) {
  cout << "Dame número: ";
  cin >> num;
}
```

Pide números al usuario hasta que se teclee el número 99

✓ ¿Cuándo finalizará el siguiente bucle?

```
while (num<0 && num>10) {
   ...
}
```

No entrará nunca puesto que es imposible que ambas condiciones se cumplan a la vez. Por tanto, la condición es falsa y no ejecuta el contenido del bucle

✓ ¿Cuántas veces se ejecutará el siguiente bucle?

```
char letra;

  cout << "¿Desea continuar (s/n)? ";
```

```
cin >> letra;
while (letra !='n' || letra != 'N') {
   cin >> letra;
}
```

Infinitas, puesto que al tratarse de un OR lógico, sea cual sea el valor de letra, la condición siempre es verdadera.

- ✓ ¿Cómo deberías haber escrito la condición anterior?

```
while (letra !='n' && letra != 'N') {
```

- ✓ ¿Qué problema existe en el siguiente fragmento de código?

```
valor = 0;
while (valor < 100) {
   cout << "Valor " << valor ;
   cout << "Obteniendo el siguiente número … ";
}
```

Puesto que la variable **valor** no se incrementa nunca, siempre se cumple la condición y por tanto, entra en un bucle infinito.

Ejemplo de funcionamiento que nos diga si cada número que tecleemos es positivo o negativo, y que pare cuando tecleemos el número 0, podría ser:

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

int main() {
  int numero;

  cout << "Teclea un número (0 para salir): ";
  cin >> numero;
  while (numero!=0) {
     if (numero > 0)
        cout << "Es positivo" << endl;
     else
        cout << "Es negativo" << endl;
     cout << "Teclea un número (0 para salir): ";
     cin >> numero;
  }
}
```

- ✓ ¿Qué ocurre en este ejemplo si se introduce 0 la primera vez?

Puesto que la condición es falsa, ni siquiera entra en el bloque del “while”, terminando el programa inmediatamente.

Importante: Una condición falsa se evalúa como el valor 0 y una condición verdadera como un valor distinto de cero, por tanto serían equivalentes las siguientes instrucciones.

```
while (numero != 0)   ⟺   while (numero)
```

- ✓ Escribe un programa que muestre por pantalla los números del 1 al 10, usando "`while`".

```
#include <iostream>
using namespace std;
const int KVECES=10;

int main() {

   int i;

   i=1;
   while (i<=KVECES){
      cout << i << endl;
      i++;
   }
}
```

Este ejemplo muestra por pantalla los números del 1 al 10, uno en cada línea.

- ✓ Implementa un programa para que dados dos enteros base y exp, -ambos no negativos y mayores que 0-, calcule la potencia del primero elevado al segundo teniendo en cuenta que no se puede utilizar el operador potencia.

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

int main() {

  float base, exp, resultado;

  cout << "Introduce la base: ";
  cin >> base;
  cout <<"Introduce el exponente: ";
  cin >> exp;

  resultado=1;
  while (exp!=0){
      resultado = resultado * base;
      exp = exp -1;
  }

  cout << "Resultado es " << resultado << endl;
}
```

Si se analiza la estructura de repetición que se ha descrito, hay un detalle que se puede haber pasado por alto en principio, ¿qué ocurre si cuando se evalúa la condición por primera vez en el bucle no se cumple la condición del mismo? La respuesta es bien sencilla, no se ejecuta una sola de las instrucciones que haya en el cuerpo de ese bucle. Por ejemplo, en el caso mostrado no podemos saber si entrará en el bucle o no puesto que desconocemos el valor de la variable *exp* ya que depende del valor introducido por el usuario.

Este detalle en ocasiones puede resultar bueno, y en otras malo, depende del diseño del algoritmo que hayamos hecho.

Es interesante, por tanto, que se pueda disponer de una estructura de repetición con toda la potencia que tienen y que, además, garantice la ejecución *al menos una vez* de las sentencias del cuerpo del bucle. Este tipo de sentencias existen, y se denominan *de repetición con condición final*, ya que la condición de permanencia o salida del bucle se evalúa después de ejecutar las sentencias del cuerpo del mismo.

Repetición con condición final.

Puede ser que en ocasiones interese plantear una estructura de repetición garantizando que las instrucciones a repetir se ejecutan al menos una vez. En tales casos se debe recurrir a las estructuras de *repetición con condición final*.

La sintaxis es la siguiente:

```
do
      S;
while (condición);
```

Al igual que ocurría con el caso anterior (instrucción `while`), si queremos que se repitan varias órdenes (muy habitual) deberemos encerrarlas entre llaves.

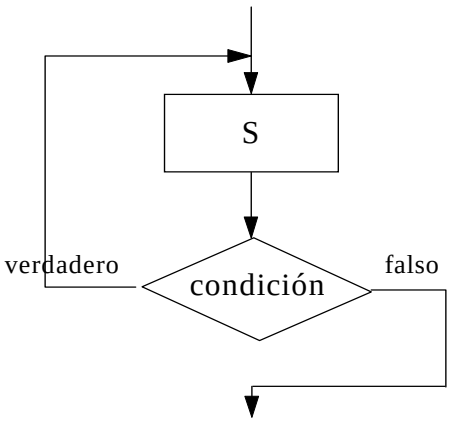
```
do {
      S;
} while (condición);
```

Donde **condición** representa una expresión booleana que:

- Si es **cierta** indica que el conjunto de instrucciones denotadas por *S* debe volver a ejecutarse de nuevo.
- Si es **falsa** indica que el conjunto de instrucciones denotadas por *S* **no** debe volver a ejecutarse de nuevo, y el flujo de control del programa se transfiere a la sentencia inmediatamente siguiente a la llave de cierre

Como se ve claramente, al evaluar la primera vez la condición del bucle, ya se ha pasado una vez por -y por tanto ejecutado- las instrucciones del cuerpo del bucle.

El *dfd* asociado a esta sentencia de repetición se puede ver en la siguiente figura:



✓ Realiza un programa que nos pide una clave de acceso (numérica) y no nos deja entrar hasta que tecleemos la clave correcta.

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

const int kvalida = 711;  // fijamos este número como clave válida

int main() {
```

```
  int clave;

  do {
    cout << "Introduzca su clave numérica: ";
    cin >> clave;
    if (clave != kvalida)
        cout << "No válida!" << endl;
  } while (clave != kvalida);
  cout << "Aceptada." << endl;
}
```

✓ Modifica el programa anterior para que sólo permita tres intentos para acertar la clave correcta.

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

const int kvalida = 711;  // fijamos este número como clave válida
const int kintentos = 3; // fijamos en 3 el número de intentos

int main() {
  int clave, intento;

  intento = 0;
  do {
    cout << "Introduzca su clave numérica: ";
    cin >> clave;
    intento++;
    if (clave != kvalida)
        cout << "No válida!" << endl;
  } while (clave != kvalida && intento < kintentos);
  if (clave == kvalida)
     cout << "Aceptada." << endl;
 else
    cout << "Rechazada. Agotaste los intentos" << endl;
}
```

Ejemplos de funcionamiento: Diseñar un algoritmo para que dados dos enteros `p' e `q', -ambos no negativos y mayores que 0-, calcule el producto del primero por el segundo haciendo uso de la suma de enteros.

Se puede plantear el producto de un número por otro, como la suma del primero tantas veces como indique el segundo:

```
algoritmo Calcular
   resultado := p + p + ... + p
falgoritmo.
```

Manera de resolverlo:

1. El cálculo del producto se irá almacenando de forma consecutiva en la variable *resultado*. Puesto que lo que se va a guardar en ella es el resultado de una suma, el valor inicial de esta variable será 0.

2. En cada pasada por el bucle, al contenido actual de la variable *resultado* se le añadirá el valor del primer término del producto, en este caso *p*.

3. Todo este proceso terminará cuando se haya dado tantas pasadas por el bucle como indique el valor del segundo término del producto, en este caso q.

Por tanto, el algoritmo **producto** puede quedar como sigue:

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

```
int main() {

   int p, q, resultado;

   cout << "Introduce el valor p: ";
   cin >> p;
   cout << "Introduce el valor q: ";
   cin >> q;

   resultado=0;
   do{
      resultado = resultado + p;
      q--;
   }while (q!=0);
}
```

Dejando a un lado los ejemplos, y si se estudia un poco más en detalle la función que realiza la **condición** de esta estructura de repetición, se ve que lo que realmente está haciendo es controlar la salida del bucle puesto que en el momento en el que su valor de verdad sea falso, entonces es cuando se produce la salida.

- ✓ ¿Qué hace el siguiente fragmento de código?

```
int x;

do {
    cin >> x;
    cout << x << endl;
}while (x!=0);
```

Recoge y muestra valores hasta que introduzcan un 0.

- ✓ ¿Muestra el 0?

Sí, porque lo comprueba después de mostrarlo.

- ✓ ¿Cómo se puede resolver manteniendo la estructura **do-while**?

```
int x;

do {
    cin >> x;
    if (x!=0)
       cout << x << endl;
}while (x!=0);
```

- ✓ ¿Cómo se puede resolver utilizando la estructura **while**?

```
int x;

cin >> x;
while (x!=0){
  cout << x << endl;
        cin >> x;
};
```

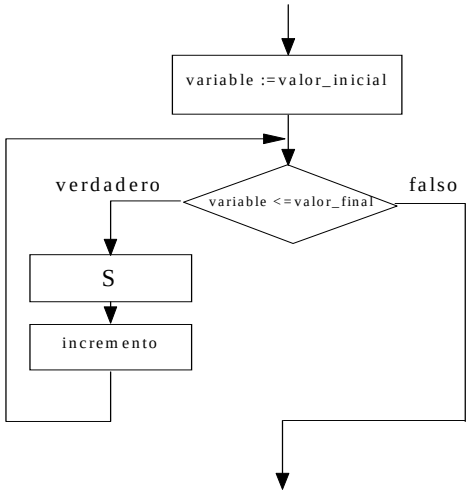
Repetición con contador.

La orden que veremos a continuación se utiliza habitualmente para crear partes del programa que **se repitan un cierto número de veces**.

La sintaxis de esta sentencia es:

```
for (inicialización contador; expresión lógica; actualización contador){
      S;
}
```

Donde en la inicialización se da valor inicial a la variable que controla el bucle. Esta variable comienza valiendo **valor_inicial**, y al finalizar cada pasada por el bucle, es decir, al término de la ejecución de las sentencias *S*, se incrementa o decrementa de la manera que se indique en la actualización del contador, así hasta alcanzar o superar -por arriba o por abajo- el **valor_final**. La figura siguiente expresa el dfd asociado con un bucle for en el que se produce un incremento de la variable que controla el bucle tras cada pasada del bucle. Cuando el valor de la variable supera al valor final, el flujo de programa se transfiere a la estructura que venga a continuación de la llave de cierre.



✓ Realiza un programa que muestre por pantalla los números del 1 al 10.

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

int main(){
  int contador;

  for (contador=1; contador<=10; contador++)
     cout << contador << endl;
}
```

Ejemplo de su uso para calcular el factorial de un número:

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

int main(){

   int i, n, factorial;

   cout << "Introduce el valor de n: ";
   cin >> n;

   factorial = 1;
   for (i=1; i<=n; i++)
      factorial = factorial * i;
}
```

C permite gran flexibilidad en este tipo de bucles pudiendo por ejemplo emplear dos o más variables de control. Por ejemplo, el bucle

```
...
  for (m=1, n=5; m+n<10; m++, n++)
      cout << "m  vale "<< m << " y n vale " << n << endl;
...
```

produce la salida:

```
m vale 1 y n vale 5
m vale 2 y n vale 6
```

Cuando se realiza la última iteración, la variable m se pone a valor 3 y la variable n a valor 7, con lo que la suma de ambas vale 10 y ya no se cumple la condición de permanencia en el bucle.

Bucles anidados

En muchas ocasiones resulta muy útil utilizar dos bucles de manera que uno está dentro del cuerpo del otro. Por ejemplo:

```
...
  for (i=1;i<=5; i++){
    for (j=1; j<=7; j++)
       cout << "*";
    cout << endl;
}
...
```

La estructura anterior dibuja un rectángulo de 5 filas y 7 columnas formando por asteriscos. El funcionamiento es:

1. Se entra en el bucle externo (controlado por i), la variable i se pone a valor 1.
2. Se entra en el bucle interno (controlado por j), la variable j se pone a valor 1. Puesto que se cumple la condición de permanencia en el bucle se dibuja un asterisco. A continuación, la variable j se incrementa a valor 2 y se vuelve a entrar en el bucle. En cada iteración de este bucle, se dibuja un asterisco por lo que se dibuja una fila de 7 asteriscos. Cuando la variable j alcanza el valor 8, ya no se cumple la condición de permanencia por lo que no entra en el bucle y el flujo de control pasa a la siguiente instrucción (`cout << endl`).
3. Esta instrucción produce un salto de línea y puesto que es la última instrucción del bucle externo, se produce el incremento de la variable i (se pone a valor 2) y se vuelve a entrar en el cuerpo del bucle. A continuación, se repite el paso 2.
4. El bucle externo realiza 5 iteraciones, lo que se corresponde con las 5 filas del rectángulo.

Bucles infinitos

Hay que ser cuidadoso al expresar la condición de permanencia en el bucle ya que se puede entrar en un bucle infinito. Por ejemplo:

```
...
```

```
n=0;
while (n<=9){
    n = n * 3;
    cout << n;
}
...
```

El bucle no termina nunca ya que la variable n siempre vale 0 y por tanto nunca supera el valor 9.

✓ Realiza un programa que muestre la tabla de multiplicar del 5.

```
#include <iostream>
using namespace std;
int main() {
   int numero;

   for (numero=0; numero<=10; numero++)
      cout << " 5 x " << numero << " = " << 5*numero << endl;
}
```

✓ Modifica el programa anterior para que muestre las tablas de multiplicar del 0 al 9.

```
#include <iostream>
using namespace std;
int main() {
  int tabla, numero;

  for (tabla=0; tabla<=9; tabla++) {
    cout << "Tabla del " << tabla << endl;
    for (numero=0; numero<=10; numero++)
       cout << tabla << " * " << numero << " = " << tabla*numero <<
endl;
    cout << " --------------- " << endl;
  }
}
```

Para “contar” no es obligatorio utilizar números, también podemos contar utilizando letras. Por ejemplo, para mostrar el alfabeto desde la ‘a’ a la ‘z’ haríamos:

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

int main() {
   char letra;

   for (letra='a'; letra<='z'; letra++)
      cout << letra << “ “ ;
   cout << endl;
}
```

Criterios para la elección de una sentencia iterativa

- Si el bucle está controlado por contador simple usar un *for*
- Si el bucle está controlado por una condición y se quiere que siempre se ejecute por lo menos una vez usar un *do-while*
- Cuando haya dudas usar un *while* ya que es el bucle más general

- ✓ ¿Qué escribiría en pantalla este fragmento de código?

```
for (i=1; i<4; i++)
      cout << i << endl;
```

Escribe los números del 1 al 3 (se empieza en 1 y se repite mientras sea menor que 4)

- ✓ ¿Qué escribiría en pantalla este fragmento de código?

```
for (i=1; i>4; i++)
      cout << i << endl;
```

No escribiría nada, porque la condición es falsa (no se cumple) desde el principio.

- ✓ ¿Qué escribiría en pantalla este fragmento de código?

```
for (i=1; i<=4; i++);
      cout << i << endl;
```

Escribe un 5, porque hay un punto y coma después del "for", de modo que repite cuatro veces una orden vacía, y cuando termina, "i" ya tiene el valor 5.

- ✓ ¿Qué escribiría en pantalla este fragmento de código?

```
for (i=1; i<4; )
      cout << i;
```

Escribe "1" continuamente, porque no aumentamos el valor de "i", luego nunca se llegará a cumplir la condición de salida.

- ✓ ¿Qué escribiría en pantalla este fragmento de código?

```
for (i=1; ; i++)
      cout << i;
```

Escribe números continuamente, comenzando en uno y aumentando una unidad en cada pasada, pero sin terminar nunca. Esto se produce porque no tiene condición de finalización.

- ✓ ¿Qué escribiría en pantalla este fragmento de código?

```
for (i= 0 ; i<= 4 ; i++) {
      if ( i == 2 )
            cout << i;
 }
```

Escribe el 2. Este se produce porque el cout está condicionado a que se cumpla la condición del if (i==2).

- ✓ ¿Cuál es el valor de **x** cuando se termina el siguiente fragmento de código?

```
for (x=0; x<100; x++);
```

x valdrá 100

- ✓ ¿Cuál es el valor de **contador** cuando se termina el siguiente fragmento de código?

```
for (contador=2; contador<10; contador+=3);
```

contador valdrá 11

- ✓ ¿Cuántas X se imprimen en pantalla?

```
for (f=0; f<10; f++);
  for (c=5; c>0; c--);
      cout << "X";

Se imprime una X
```

- ✓ ¿Cuántas X se imprimen en pantalla?

```
for (f=0; f<10; f++)
  for (c=5; c>0; c--);
      cout << "X";

Se imprime una X
```

- ✓ ¿Cuántas X se imprimen en pantalla?

```
for (f=0; f<10; f++);
  for (c=5; c>0; c--)
      cout << "X";

Se imprimen cinco X
```

- ✓ ¿Cuántas X se imprimen en pantalla?

```
for (f=0; f<10; f++)
  for (c=5; c>0; c--)
      cout << "X";

Se imprimen cincuenta X
```

- ✓ ¿Cuál es el error en este fragmento de código?

```
for (contador = 1; contador < kmax; contador++);
   cout << "Contador = " << contador << endl;

Mostrará por pantalla:
  Contador = 5
El error es que la instrucción de cout no está dentro del for
porque en la instrucción for se ha escrito un ;
Si se elimina el ; ya es correcto y mostraría:
  Contador = 1
  Contador = 2
  Contador = 3
  Contador = 4
```