Sentencias de control. Estructuras de selección

Las estructuras de selección permiten realizar distintas acciones en el programa dependiendo del valor de una condición.

Selección simple.

La estructura alternativa permite seleccionar qué instrucciones se han de ejecutar dependiendo del valor que tengan determinadas variables en un momento dado.

La sintaxis que tiene la *alternativa* es la siguiente:

```
if (condición){
        S;
}
```

Donde **condición** representa una expresión booleana y S la secuencia es un conjunto de instrucciones que solo se ejecutarán si la condición booleana *condición* se evalúa a *cierto*.

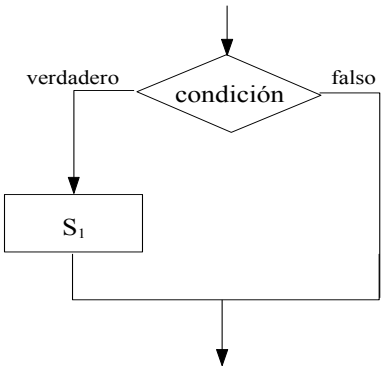
Cuando la secuencia de instrucciones está formada por una única sentencia, no es necesario el uso de llaves. Sin embargo, si la secuencia de instrucciones es compleja, el conjunto debe ir entre llaves.

Los operadores que se pueden emplear en una expresión booleana con el fin de obtener los valores *cierto* y *falso* son los siguientes:

Lógicos: **&&** **||** **!**

Relacionales: **<** **<=** **==** **<>** **>=** **>**

En la siguiente figura se puede ver cuál sería la representación mediante un *diagrama de flujo* de esta sentencia.



Los diagramas de flujo de datos -*dfd*- realizan una representación gráfica de los algoritmos. Como su nombre indica, en ellos se resalta, principalmente, la lógica o el flujo de control del programa, es decir, lo que se hace en cada momento.

Las expresiones condicionales se evalúan, en lo que se denomina, modo '*short-circuit*', es decir, dadas expresiones booleanas como:

- "(condición1 || condición2)'', si *condición1* es cierta, entonces no se evalúa *condición2*, ya que el valor de verdad de la expresión será cierto, valga lo que valga *condición2*.
- "(condición1 && condición2)'', si *condición1* es falsa, entonces no se evalúa *condición2*, ya que el valor de verdad de la expresión será falso, valga lo que valga *condición2*.

La ventaja de este modo de evaluación de las expresiones booleanas radica en el incremento de velocidad de la ejecución de los programas, ya que no se invierte tiempo haciendo cálculos que no van a influir en el valor de verdad final de la expresión con la que estamos tratando. Sin embargo, plantea ciertas pegas a los programadores a través de lo que se conocen como *efectos laterales* o *side-effects*, es decir, trozos de código que el programador esperaba que fuesen ejecutados, no llegan a serlo debido a la forma de evaluar la expresión.

Es importante saber que no todos los lenguajes de programación evalúan las expresiones booleanas de este modo (El lenguaje C sí lo hace, mientras que otros lenguajes no lo hacen). El usuario de un lenguaje de programación debe conocer el modo de evaluación de las expresiones booleanas que sigue el lenguaje elegido para programar, ya que esto puede repercutir en la correcta ejecución de los algoritmos debido los efectos laterales.

✓ ¿Qué hace el siguiente fragmento de código?

```
int num;

cout << "Introduce un número";
cin >> num;
if (num>0)
   cout << "El número es positivo";
```

Pide un número al usuario y si es distinto de cero muestra el mensaje en pantalla.

Ejemplo. Expresar mediante sentencias condicionales que:

q = 1 si (n > 0)

0 si (n = 0)

-1 si (n < 0)

```
#include <iostream>

using namespace std;
int main() {
   int q, n;

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

   if (n>0)
      q = 1;
   if (n==0)
      q = 0;
   if (n<0)
      q = -1;
}
```

✓ ¿Qué hace el siguiente fragmento de código?

```
int num1, num2;
```

```
cout << "Introduce dos números";
cin >> num1 >> num2;
if (num1%num2==0)
   cout << "OK";
```

Pide dos números al usuario y comprueba si el primero es múltiplo del segundo.

✓ ¿Y si queremos que compruebe si cualquiera de los dos números es múltiplo del otro?

Modificamos la condición de esta manera:

```
if (num1%num2==0 || num2%num1==0)
```

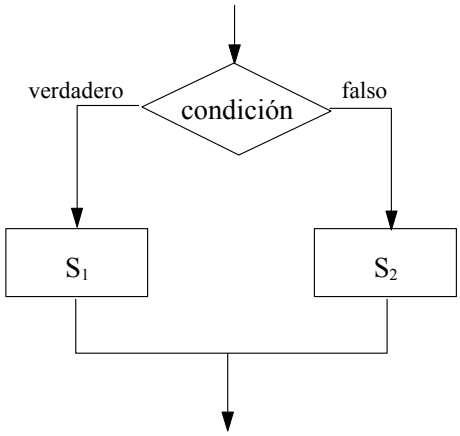
Selección doble.

La estructura alternativa permite un enunciado más general, el cual da la posibilidad de expresar una serie de sentencias que se ejecutan si no se cumple la condición. La sintaxis es:

```
if (condición)
   S1;
else
   S2
```

Donde, *condición* representa una expresión booleana, y S_1 y S_2 representan conjuntos de instrucciones. Las primeras solo se ejecutarán si la condición booleana *condición* se evalúa a *cierto*, y las segundas si se evalúa a *falso*.

La representación mediante un *dfd* de esta sentencia se puede ver en la siguiente figura.



Al igual que con la selección simple, las secuencias de instrucciones deben ir entre llaves cuando se trate de más de una sentencia.

✓ ¿Qué hace este fragmento de código?

```
…
if ((car >= 'A' && car <= 'Z') || (car >= 'a' && car <='z')
   cout << "Es una letra"
else
   cout << "No es una letra"
```

Comprueba si el contenido de la variable car (que será de tipo char) es una letra o es algo distinto de una letra (por

ejemplo un signo de puntuación, un dígito o cualquier otro carácter)

✓ ¿Qué muestra en pantalla este fragmento de código?

```
int x = 0;

if (x=5)
   cout << "SI";
else
   cout << "NO";
```

La condición del if es una asignación, no una comparación (para comparar valores usamos el operador ==). Por tanto, lo que hay escrito en la condición se evalúa como verdadero (distinto de 0) y se escribe SI

✓ ¿Qué muestra en pantalla este fragmento de código?

```
int n = 5;
if (n== 1 || 3)
   cout << "SI";
else
   cout << "NO";
```

SI, la condición no está escrita de manera correcta. La forma correcta de comprobar si la variable n vale 1 ó 3 es: `if (n== 1 || n==3)`

✓ ¿Qué muestra en pantalla este fragmento de código?

```
int n = 10;
if (n%2==0 && n<10)
   cout << "Es un número par menor que 10";
else
   cout << "No es un número par menor que 10";
```

n vale 10, la condición se evalúa a falsa ya que aunque es un número par (`n%2==0` es verdadero), n no es menor que 10. Por tanto se escribe `No es un número par menor que 10.`

¿Cómo funciona realmente la condición de un if?

Una condición cuyo resultado sea "falso" devuelve un 0 y otra cuyo resultado sea "verdadero" devuelve un valor distinto de 0. En general, si la condición es algo que vale 0, se considera que es falsa (no se cumple) y si es algo distinto de cero se considera que es verdadera (sí se cumple). Esto permite hacer cosas como esta:

```
int n;

cout << "Introduce un número";
cin >> n;
if (n)  //equivalente a if (n!=0)
   cout << "El número no es cero.\n";
```

✓ ¿Qué muestra en pantalla este fragmento de código?

```
int n = 10;

if (n)
   cout << "SI"
else
   cout << "NO"
```

Si la condición del if es algo que vale distinto de 0 se considera que la condición es verdadera (se cumple). Por tanto escribe SI

Selección múltiple.

`if-else` anidada

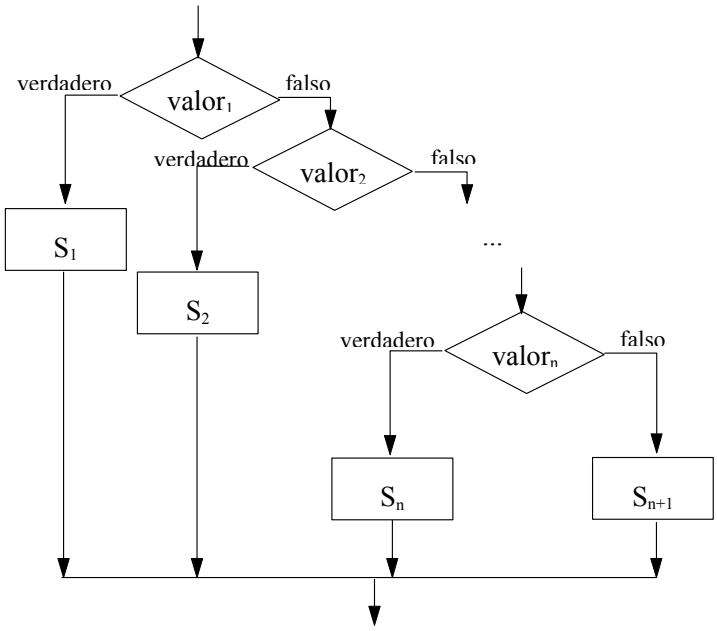
Consiste en usar estructuras if-else de manera **anidada,** es decir, unas estructuras if-else dependientes de otras. La sintaxis es:

```
if (expresión_lógica_1) {
        S1;
} else if (expresión_lógica_2) {
              S2;
        } else if (expresión_lógica_3) {
                    S3;
                }
```

Donde S_1, S_2 y S_3, corresponden a conjuntos de instrucciones.

Sólo se ejecuta la primera de las secuencias de sentencias cuya expresión lógica asociada se evalúe a verdadero. Si todas las expresiones lógicas se evalúan a falso, entonces se ejecuta la secuencia de sentencias situada a continuación del `if-else` anidado a menos que la última alternativa este asociada a una parte `else`, en cuyo caso se ejecutará esta rama.

La siguiente figura representa el dfd.



Ejemplo

Implementa un algoritmo en C que pida al usuario la edad de una persona y muestre un mensaje indicando si es menor de edad, adulto o anciano.

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

int main() {
   int edad;

   cout << "Introduce la edad: ";
   cin >> edad;
   if (edad<18)
      cout << "Es menor de edad" << endl;
   else if (edad <70)
         cout << "Es adulto" << endl;
      else
         cout << "Es anciano" << endl;
}
```

switch

En ocasiones es interesante poder distinguir entre diversas alternativas sin tener que emplear múltiples sentencias `if...else`, las cuales producen el efecto deseado, pero oscurecen el código escrito. En estos casos podemos usar la sentencia `switch`, la cual tiene la siguiente sintaxis:

```
switch (expresión) {
    valor_1:  S1;
              break;
    valor_2:  S2;
              break;
    valor_n:  Sn;
              break;
    default:  Sn+1;
}
```

Como se observa, se pregunta por el valor actual de la expresión, y dependiendo de cuál sea éste, se selecciona una u otra secuencia de instrucciones S_i a ejecutar. Si ocurre que el valor de la expresión no se corresponde con ninguno de los valores '**valor**' que esperábamos, entonces se ejecuta la secuencia de instrucciones asociadas a la palabra reservada **`default`**, la cual representa cualquier otro valor de la expresión no tenido en cuenta previamente por la sentencia **`switch`**. Una vez ejecutada esta secuencia de instrucciones, el flujo del programa se transmite automáticamente al final de la sentencia **`switch`**, es decir, a la primera sentencia que siga a la llave final.

El tipo de la expresión debe ser escalar (entero, carácter, booleano). Las etiquetas case no pueden estar repetidas y deben ser del mismo tipo que el selector (expresión). Cada sentencia debe estar precedida por una o más etiquetas **`case`**. La sentencia **`break`** hace que se transfiera el control fuera de la estructura **`switch`**. En caso de no existir break, una vez encontrada la etiqueta case que coincide con el valor del selector, se ejecutarían la sentencia asociada a la etiqueta en cuestión y todas las sentencias que hay hasta el final del **switch**.

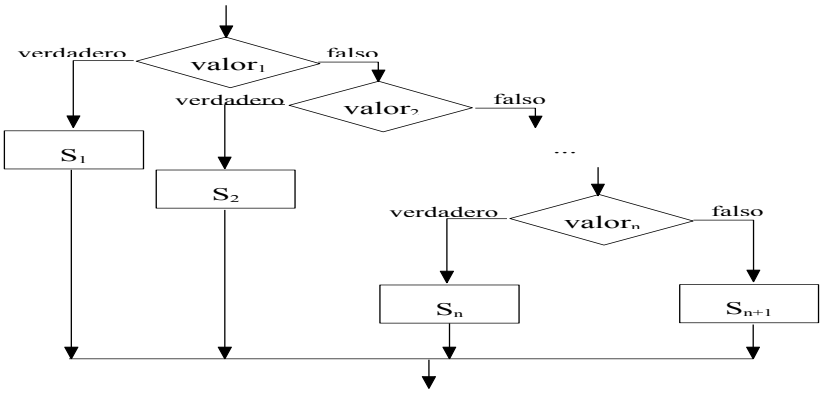
- ✓ ¿Qué hace este fragmento de código?

```
int n;

cout << "Introduce un número entre 1 y 5";
cin >> n;

switch (n){
   case 1:
```

```
        case 3:
        case 5: cout << "El número es impar";
                break;
        case 2:
        case 4: cout << "El número es par";
                break;
        default: cout << "ERROR: No es un número entre 1 y 5";
    }
```

```
Pide un número al usuario y si n tiene los valores 1, 3 ó 5
muestra un mensaje indicando que es impar, si vale 2 ó 4
indica que es par y si vale un número que no está en el
intervalo 1..5, muestra un mensaje de error.
```

La representación mediante un *dfd* de esta sentencia la podemos ver en la siguiente figura



Un ejemplo de uso habitual es la selección de una opción de un menú.

Ejemplo

Implementa un algoritmo en C que realice las cuatro operaciones básicas de una calculadora (suma, resta, multiplicación y división). El programa debe leer los dos números y la operación y devolver el resultado.

Cuando se realiza una división siempre hay que controlar que el divisor sea distinto de 0, es por ello que el case correspondiente a la división tiene un if para controlar esta situación.

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

int main() {
   float a, b, resultado;
   char op;

   cout << "Introduce el primer término: ";
   cin >> a;
   cout << "Introduce el segundo término: ";
   cin >> b;
   cout << "Introduce la operación ";
   cin >> op;

   switch (op){
      case '+': resultado = a + b;
                break;
```

```
        case '-': resultado = a - b;
                  break;
        case '*': resultado = a * b;
                  break;
        case '/': if (b!=0)
                      resultado = a / b;
                  else
                      resultado=-1;
                  break;
    }
   if (resultado!=-1)
        cout << "El resultado es: " << resultado << endl;
   else
        cout << "No se puede dividir entre 0” << endl;
}
```