Curso Básico de C++

IDE Visual C++ Express Edition 2008

(II)

ESTRUCTURAS

En general, si se han de ejecutar varias sentencias (más de una), se deben encerrar entre llaves "{" y "}", si sólo se trata de una única sentencia, estas llaves se pueden omitir.

Estructuras de sentencias

Estructuras de selección: Permite seleccionar qué sentencias se ejecutarán según sea una expresión. Permiten controlar el flujo del programa, seleccionando distintas sentencias en función de diferentes circunstancias.

Sentencia **if...else**, permite la ejecución condicional de una/varias sentencia/s. Sintaxis:

```
if (<condición>) <sentencia1>
  [else <sentencia2>]
```

Si la condición es verdadera se ejecutará la sentencia1, si es falsa, (y si existe la parte del **else**), se ejecutará la sentencia2. La claúsula **else** es opcional.

Ejemplo: Un programa para que pida por pantalla dos números, a y b y me presente cual de los dos es el mayor (o si son iguales).

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

void main(){
int a,b;

printf("Introduce un número a: ");
scanf("%i",&a);
printf("\nIntroduce un número b: ");
scanf("%i",&b);

if (a>b)
      printf("\nEl mayor es %i",a);
else
    if (b>a)
       printf("\nEl mayor es %i",b);
      else
         printf("\nLos dos son iguales %i y %i",a,b);

printf("\n\nPulsa cualquier tecla para terminar....");
tecla = getch();

}
```

Para "capturar" o recoger un valor desde el teclado, usaremos la función **scanf()**, entre comillas, ponemos el código **%** seguido del tipo de dato y separado por coma, el nombre de la variable donde recogeremos el valor escrito.

Fijaos en esto, para "capturar" desde el teclado un valor numérico o de letra, debemos hacerlo anteponiendo el símbolo **&** al nombre de la variable, en caso contrario, no se captura bien.

Sentencia switch, esta sentencia es una generalización de las sentencias **if...else**. En el caso de las sentencias **if**, la expresión que se evalúa como condición es booleana, lo que quiere decir que sólo hay dos valores posibles, y por lo tanto, sólo se puede elegir entre dos sentencias a ejecutar.

En el caso de la sentencia **switch**, la expresión a evaluar será entera, por lo tanto, el número de opciones es mucho mayor, y en consecuencia, también es mayor el número de diferentes sentencias que se pueden ejecutar. Sintaxis:

```
switch (<expresión entera>)
{
   [case <expresión_constante1>: [<sentencias1>]]
   [case <expresión_constante2>: [<sentencias2>]]
   ...
   [case <expresión_constantenn>: [<sentenciasnn>]]
   [default : [<sentencia>]]
}
```

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 expresión entera evaluada dentro del **switch**. A partir de ese punto todas las sentencias serán ejecutadas hasta el final del **switch**, es decir hasta llegar al "}". Esto es así porque las etiquetas sólo marcan los puntos de entrada después de una ruptura de la secuencia de ejecución, pero no marcan los puntos de salida.

Esta estructura está diseñada para ejecutar cierta secuencia de instrucciones, empezando a partir de un punto diferente, en función de un valor entero y dejando sin ejecutar las anteriores a ese punto.

Esta característica también nos permite ejecutar las mismas sentencias para varias etiquetas distintas, y en el apartado siguiente veremos (aunque vamos a adelantarlo ahora) que se puede eludir la ejecución secuencial normal usando la sentencia de ruptura **break** para ejecutar sólo parte de las sentencias.

Si el valor de la expresión entera no satisface ningún **case**, es decir, si no existe un **case** con una expresión constante igual al valor de la expresión entera, el control parará a la sentencia etiquetada con la etiqueta **default**.

Todas las etiquetas son opcionales, tanto **default** como todos los **case**. Si no se satisface ningún **case**, ni aparece la etiqueta **default**, se abandonará la sentencia **switch** sin ejecutar ninguna sentencia.

Resumiendo, que os leo el pensamiento y se que no os habéis enterado de nada, switch lo usamos ….

Si queremos "evaluar" una variable de tipo entero, es decir, un número o una letra (carácter, ya que un solo carácter se entiende como un entero sin signo de 0 a 255), de tal manera, que dependiendo del valor que tenga dicha variable, se "ejecuten" una serie de sentencias.

Ejemplo: Un programa para que pida por pantalla una letra y me presente si es o no una vocal.

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

void main(){
bool EsVocal;
char letra;
printf("Introduce una letra: ");
scanf("%c",&letra);
switch(letra)
{
   case 'a': // como no ponemos break, saltaremos al case 'e':
   case 'e': // como no ponemos break, saltaremos al case 'i':
   case 'i': // como no ponemos break, saltaremos al case 'o':
   case 'o': // como no ponemos break, saltaremos al case 'u':
   case 'u':
      EsVocal = true;
      break; // si no ponemos esto, se saltará al default
   default:  // si no se ha entrado por ninguna de las anteriores, siempre
irá al default
      EsVocal = false;
}
if (EsVocal==true)
    printf("Es una vocal");
else
      printf("No es una vocal");


printf("\n\nPulsa cualquier tecla para terminar....");
tecla = getch();

}
```

break, la sentencia de ruptura.

Como habéis visto, esta palabra "break", es decir, cascar o romper, la usamos cuando no me interesa que se siga ejecutando las sentencias que se encuentren a continuación. Nos será especialmente útil en las siguientes estructuras llamadas de "bucle" junto a otra sentencia llamada **continue**.

Estructuras de repetición (bucles): Estos tipos de sentencias son el núcleo de cualquier lenguaje de programación, y están presentes en la mayor parte de ellos. Nos permiten realizar tareas repetitivas, y se usan en la resolución de la mayor parte de los problemas.

Un pequeña reseña histórica

El descubrimiento de los bucles se lo debemos a Ada Byron, así como el de las subrutina (que no es otra cosa que una función o procedimiento). Está considerada como la primera programadora, y ella misma se autodenominaba analista, lo que no deja de ser sorprendente, ya que el primer ordenador no se construyó hasta un siglo después.

Generalmente estas sentencias tienen correspondencia con estructuras de control equivalentes en *pseudocódigo*. El *pseudocódigo* es un lenguaje creado para expresar *algoritmos* formalmente y de manera clara. No es en si mismo un lenguaje de programación, sino más bien, un lenguaje formal (con reglas muy estrictas), pero humano, que intenta evitar ambigüedades.

A su vez, un *algoritmo* es un conjunto de reglas sencillas, que aplicadas en un orden determinado, permiten resolver un problema más o menos complejo.

Por ejemplo, un *algoritmo* para saber si un número *N*, es primo, puede ser:

Cualquier número es primo si sólo es divisible entre si mismo y la unidad.

Por lo tanto, para saber si un número *N* es primo o no, bastará con dividirlo por todos los números entre 2 y *N-1*, y si ninguna división es exacta, entonces el número *N* es primo.

Pero hay algunas mejoras que podemos aplicar:

La primera es que no es necesario probar con todos los números entre 2 y *N-1*, ya que podemos dar por supuesto que si *N* no es divisible entre 2, tampoco lo será para ningún otro número par: 4, 6, 8..., por lo tanto, después de probar con 2 pasaremos al 3, y después podemos probar sólo con los impares.

La segunda es que tampoco es necesario llegar hasta *N-1*, en realidad, sólo necesitamos llegar hasta el valor entero más cercano a la raíz cuadrada de *N*.

Esto es así porque estamos probando con todos los números menores que *N* uno a uno. Supongamos que vamos a probar con un número *M* mayor que la raíz cuadrada de *N*. Para que *M* pudiera ser un divisor de *N* debería existir un número *X* que multiplicado por *M* fuese igual a *N*.

$N = M \times X$

El caso extremo, es aquel en el que *M* fuese exactamente la raíz cuadrada de *N*. En ese caso, el valor de *X* sería exactamente *M*, ya que ese es el valor de la raíz cuadrada de *N*:

$N = M^2 = M \times M$

Pero en el caso de que *M* fuese mayor que la raíz cuadrada de *N*, entonces el valor de *X* debería ser menor que la raíz cuadrada de *N*. Y el caso es que ya hemos probado con todos los números menores que la raíz cuadrada de *N*, y ninguno es un divisor de *N*.

Por lo tanto, ningún número *M* mayor que la raíz cuadrada de *N* puede ser divisor de *N* si no existen números menores que la raíz cuadrada de *N* que lo sean.

El pseudocódigo para este algoritmo sería algo parecido a esto:

```
¿Es N=1? -> N es primo, salir
¿Es N=2? -> N es primo, salir
Asignar a M el valor 2
mientras M <= raíz cuadrada(N) hacer:
   ¿Es N divisible entre M? -> N no es primo, salir
   Si M=2 entondes Asignar a M el valor 3
   Si M distinto de 2 entonces Asignar a M el valor M+2
Fin del mientras
N es primo ->salir
```

Bucles "**mientras**"

Es la sentencia de bucle más sencilla, y sin embargo es tremendamente potente. En C++ se usa la palabra reservada **while** (que significa "mientras"), y la sintaxis es la siguiente:

while (<condición>) <sentencia>

La sentencia es ejecutada repetidamente *mientras* la condición sea verdadera. Si no se especifica condición se asume que es **true**, y el bucle se ejecutará indefinidamente. Si la primera vez que se evalúa la condición resulta falsa, la sentencia no se ejecutará ninguna vez.

Las condiciones no son otra cosa que expresiones de tipo booleano, cualquier otro tipo de expresión se convertirá a tipo booleano, si es posible. Y si no lo es, se producirá un error.

Por ejemplo:

```
while (x < 100) x = x + 1;
```

El while ejecuta una única sentencia (no hacen falta llaves).

Se incrementará el valor de x mientras x sea menor que 100.

Este ejemplo puede escribirse, usando el C++ con propiedad y elegancia (es decir, *con clase*), de un modo más compacto:

```
while (x++ < 100);
```

El while no ejecuta ninguna sentencia (no hacen falta llaves).

Fijaos, que `x++`, hace que primero se evalúe la condición con el valor que tuviera al entrar en la sentencia `while` y después se incrementa en una unidad, se vuelve a evaluar la condición hasta que el incremento sea = a 100, que hará que termine y salga.

Bucle "**hacer...mientras**"

Esta sentencia va un paso más allá que el **while**. La sintaxis es la siguiente:

```
do <sentencia> while(<condicion>);
```

La sentencia es ejecutada repetidamente mientras la condición resulte verdadera. Si no se especifica condición se asume que es **true**, y el bucle se ejecutará indefinidamente igual que en el `while`. Pero a diferencia del bucle **while**, la evaluación de la condición se realiza después de ejecutar la sentencia, de modo que ésta se ejecutará al menos una vez. Por ejemplo:

```
do
   x = x + 1;
while (x < 100);
```

En este bucle se incrementará el valor de x hasta que valga 100.

Pero aunque la condición sea falsa, por ejemplo, si x vale inicialmente 200, la sentencia `x = x + 1;`, se ejecuta primero, y después se verifica la condición.

Bucle "**para**"

Por último el bucle *para*, que usa la palabra reservada **for**. Este tipo de bucle es el más elaborado. La sintaxis es:

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

La sentencia es ejecutada repetidamente mientras la condición resulte verdadera, o expresado de otro modo, hasta que la evaluación de la condición resulte falsa.

Antes de la primera iteración se ejecutará la iniciación del bucle, que puede ser una expresión o una declaración. En este apartado se suelen iniciar las variables usadas en el bucle. Estas variables también pueden ser declaradas en este punto, pero en ese caso tendrán validez (ámbito) sólo dentro del bucle **for**.

Después de cada iteración se ejecutará el incremento de las variables del bucle. Este incremento también es una sentencia de asignación, y no tiene por qué ser necesariamente un incremento.

En general, podemos considerar que la parte de *inicialización* establece las condiciones iniciales del bucle, la parte de la *condición* establece la condición de salida, y la parte del *incremento*, modifica las condiciones iniciales para establecer las de la siguiente iteración del bucle, o para alcanzar la condición de salida.

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

```
for(int i = 0; i < 100; i = i + 1) { printf("%i\n",i);}
for(int i = 100; i < 0; i = i - 1) { printf("%i\n",i);}
```

Un ejemplo más completo, usando dos estructuras junto con el operador matemático %, que devuelve el resto de una división entera, podría ser un programa para que presente por pantalla los números impares que estén entre los primeros 100 números naturales.

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

void main(){
int i; // Variable que usaremos para “iterar” i y analizar si es o no primo

printf("\nLos primos menores a 100 son: 1"); // Mostramos 1 pues ya sabemos
                                              //que lo es
for (i=2;i<=100;i++){
      if (i%2!=0) {     // si no es divisible por 2 es impar
         if (i<100) printf(",");    //si no hemos llegado al último pongo “,”
         printf("%d",i);      //presento después de la coma el número impar
      }
}
printf("\n\nPulsa cualquier tecla para terminar....");
tecla = getch();

}
```