Curso Básico de C++

IDE Visual C++ Express Edition 2008

(III)

Bucles: Sentencia *continue*

El uso de esta sentencia dentro de un bucle ignora el resto del código de la iteración actual, y comienza con la siguiente, es decir, se transfiere la ejecución a la evaluación de la condición del bucle. Sintaxis:

```
y = 0;
x = 0;
while(x < 1000)
{
   x++;
   if(y >= 100) continue;
   y++;
}
```

En este ejemplo la línea `y++` sólo se ejecutaría mientras y sea menor que 100, en cualquier otro caso el control pasa a la siguiente iteración, con lo que la condición del bucle volvería a evaluarse.

Bucles: Sentencia *break*

El uso de esta sentencia dentro de un bucle, una sentencia de selección o de un bloque, transfiere la ejecución del programa a la primera sentencia que haya a continuación. Esto es válido para sentencias **switch**, como vimos hace un rato, pero también lo es para sentencias **while**, **do...while**, **for** e **if**.

En general, una sentencia **break** transfiere la ejecución secuencial a la siguiente sentencia, abandonando aquella en que se ejecuta.

```
y = 0;
x = 0;
while(x < 1000)
{
   if(y == 1000) break;
   y++;
}
x = 1;
```

*En este ejemplo el bucle no terminaría nunca si no fuera por la línea del **break**, ya que x no cambia. Después del **break** el programa continuaría en la línea `x = 1`.*

Salida de texto en español

Para obtener una salida de textopa pantalla, manteniendo acentos y símbolos del español, suele conseguirse añadiendo la siguiente librería: `#include <locale.h>` y escribiendo la siguiente sentencia dentro del código (justo después del `void main() ).`

```
setlocale( LC_ALL, "Spanish" );
```

Más sobre operadores

Otros dos operadores unitarios. Se trata de operadores un tanto especiales, ya que sólo pueden trabajar sobre variables, pues implican una asignación. Se trata de los operadores '++' y '--'. El primero incrementa el valor del operando y el segundo lo decrementa, ambos en una unidad.

Existen dos modalidades, dependiendo de que se use el operador en la forma de prefijo o de sufijo. Sintaxis:.

```
<variable> ++ (post-incremento)
++ <variable> (pre-incremento)
<variable>--  (post-decremento)
-- <variable> (pre-decremento)
```

En su forma de prefijo, el operador es aplicado antes de que se evalúe el resto de la expresión; en la forma de sufijo, se aplica después de que se evalúe el resto de la expresión. Veamos un ejemplo, en las siguientes expresiones "a" vale 100 y "b" vale 10:

```
c = a + ++b;
```

En este primer ejemplo primero se aplica el pre-incremento, y b valdrá 11 a continuación se evalúa la expresión "a+b", que dará como resultado 111, y por último se asignará este valor a c, que valdrá 111.

```
c = a + b++;
```

En este segundo ejemplo primero se avalúa la expresión "a+b", que dará como resultado 110, y se asignará este valor a c, que valdrá 110. Finalmente se aplica en post-incremento, y b valdrá 11.

Los operadores unitarios sufijos (post-incremento y post-decremento) se evalúan después de que se han evaluado el resto de las expresiones.

- ✓ En el primer ejemplo primero se evalúa ++b, después a+b y finalmente c =<resultado>.
- ✓ En el segundo ejemplo, primero se evalúa a+b, después c = <resultado> y finalmente b++.

Es muy importante no pensar o resolver las expresiones C como ecuaciones matemáticas, **NO SON EXPRESIONES MATEMATICAS**. No veas estas expresiones como ecuaciones, **NO SON ECUACIONES**. Esto es algo que se tarda en comprender al principio, y que después aprendes y dominas hasta el punto que no te das cuenta.

Por ejemplo, piensa en esta expresión:

```
b = b + 1;
```

Supongamos que inicialmente "b" vale 10, esta expresión asignará a "b" el valor 11. Veremos el operador "=" más adelante, pero por ahora no lo confundas con una igualdad matemática. En matemáticas la expresión anterior no tiene sentido, en programación sí lo tiene.

Operadores de asignación

Existen varios operadores de asignación, el más evidente y el más usado es el "=", pero en C++ este no es el único que existe.

Aquí hay una lista: "=", **"*=", "/=", "%=", "+=", "-="**, "<<=", ">>=", "&=", " ^=" y "|=". Y la sintaxis es:

El funcionamiento es siempre el mismo, primero se evalúa la expresión de la derecha, se aplica el operador mixto, si existe y se asigna el valor obtenido a la variable de la izquierda.

Los operadores del segundo al sexto son combinaciones del operador de asignación "=" y de los operadores aritméticos que hemos visto en el punto anterior. Tienen las mismas limitaciones que ellos, es decir, el operador "%=" sólo se puede aplicar a expresiones enteras.

El resto de los operadores son operadores de bits, y los veremos más adelante, en otro de los capítulos dedicado a operadores.

Operadores de comparación

Estos operadores comparan dos operandos, dando como resultado valores booleanos, **true** (verdadero) o **false** (falso), dependiendo de si los operandos cumplen o no la operación indicada.

Son "==" (dos signos = seguidos), "!=", "<", ">", "<=" y ">=", que comprueban relaciones de igualdad, desigualdad y comparaciones entre dos valores aritméticos. Sintaxis:

```
<expresión1> == <expresión2>
<expresión1> != <expresión2>
<expresión1> > <expresión2>
<expresión1> < <expresión2>
<expresión1> <= <expresión2>
<expresión1> >= <expresión2>
```

Si el resultado de la comparación resulta ser verdadero, se retorna **true**, en caso contrario **false**. El significado de cada operador es evidente:

== igualdad	> mayor que	>= mayor o igual que
!= desigualdad	< menor que	<= menor o igual que

En la expresión "E1 <operador> E2", los operandos tienen algunas restricciones, pero de momento nos conformaremos con que E1 y E2 sean de tipo aritmético. El resto de las restricciones las veremos cuando conozcamos los punteros y los objetos.

Operadores lógicos

Los operadores "&&", "||" y "!" relacionan expresiones lógicas, dando como salida a su vez nuevas expresiones lógicas. Sintaxis:

```
<expresión1> && <expresión2>
<expresión1> || <expresión2>
!<expresión>
```

- El operador "&&" equivale al "AND" o "Y"; devuelve **true** sólo si los dos operandos **true** o lo que es equivalente, distintas de cero. En cualquier otro caso el resultado es **false**.
- El operador "||" equivale al "OR" u "O inclusivo"; devuelve **true** si cualquiera de las expresiones evaluadas es **true**, o distinta de cero, en caso contrario devuelve **false**.
- El operador "!" es equivalente al "NOT", o "NO", y devuelve **true** cuando la expresión evaluada es **false** o cero, en caso contrario devuelve **false**.

Existen unas reglas que en muchas ocasiones nos puede resultar útil, ya que nos puede ahorrar tiempo y comprobaciones adicionales, se aplican de forma diferente a expresiones AND y OR.

1. En el caso de operaciones AND, consiste en que si la primera expresión evaluada es **false**, la segunda ni siquiera se evalúa, ya que el resultado será siempre **false** independientemente del valor del segundo operando.

2. En el caso de operaciones OR, si la primera expresión sea **true**, la segunda no se evalúa, ya que el resultado será siempre **true**, independientemente del valor de la segunda expresión.

*Esto es porque en una operación && el resultado sólo puede ser **true** cuando los dos operandos sean **true**, y en una operación || el resultado sólo puede ser **false** si ambos operandos son **false**. En el momento en que en una expresión AND uno de los operandos sea **false**, o que en una expresión OR uno de los operandos sea **true**, el valor del otro operando es irrelevante.*

Si tenemos en cuenta este comportamiento, podremos ahorrar tiempo de ejecución si colocamos en primer lugar la expresión más fácil de calcular, o aquella cuyo valor sea más probablemente **false** en el caso de una expresión AND o **true**, para una expresión OR.

También habrá casos en que una de las expresiones sea indeterminada cuando la otra sea **false** en una expresión AND, o **true** en una expresión OR. En ese caso, será preferible colocar la expresión potencialmente indeterminada en el segundo lugar.

EJERCICIOS EJEMPLO RESUELTOS

Escribir un programa que muestre una lista de números del 1 al 20, indicando a la derecha de cada uno si es divisible por 3 o no.

```
// Este programa muestra una lista de números,
// indicando para cada uno si es o no múltiplo de 3.
// © Pedro L. Recuenco Andrés

#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <locale.h>

int main() // función principal
{
      setlocale( LC_ALL, "Spanish" );

      int i; // variable para bucle

      for(i = 1; i <= 20; i++) // bucle for de 1 a 20
      {
            printf("%d", i); // muestra el número
            if(i % 3 == 0)
                  printf(" es múltiplo de 3"); // resto==0
            else
                  printf(" no es múltiplo de 3"); // resto != 0
            printf("\n"); // salto de línea
      }


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

}
```

El enunciado es el típico de un problema que puede ser solucionado con un bucle **for**. Observa el uso de los comentarios, y acostúmbrate a incluirlos en todos tus programas. Acostúmbrate también a escribir el código al mismo tiempo que los comentarios. Si lo dejas para cuando has terminado el programa, probablemente sea demasiado tarde, y la mayoría de las veces no lo harás. ;-)

También es una buena costumbre incluir al principio del programa un comentario extenso que incluya el enunciado del problema, añadiendo también el nombre del autor y la fecha en que se escribió. Además, cuando hagas revisiones, actualizaciones o correcciones deberías incluir una explicación de cada una de ellas y la fecha en que se hicieron.

Una buena documentación te ahorrará mucho tiempo y te evitará muchos dolores de cabeza.

Escribir un programa que muestre una salida de 20 líneas de este tipo:

```
1
1 2
1 2 3
1 2 3 4
...
```

Resolución

```
// Este programa muestra una lista de números
// de este tipo:
// 1
// 1 2
// 1 2 3
// ...
// © Pedro L. Recuenco Andrés


#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <locale.h>

int main() // función principal
{
   setlocale( LC_ALL, "Spanish" );

   int i, j; // variables para bucles

   for(i = 1; i <= 20; i++) // bucle hasta i igual a 20
   {
      for(j = 1; j <= i; j++) // bucle desde 1 a i
         printf("%d ",j); // muestra el número
      printf("\n"); // salto de línea
   }


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

}
```

Este ejemplo ilustra el uso de bucles anidados. El bucle interior, que usa *j* como variable toma valores entre 1 e *i*. El bucle exterior incluye, además del bucle interior, la orden de cambio de línea, de no ser así, la salida no tendría la forma deseada. Además, después de cada número se imprime un espacio en blanco, de otro modo los números aparecerían amontonados.