Tema 3

Estructuras de control en C++

Índice general

3.1. Bloques de sentencias

En C++ el concepto de bloque de sentencias se utilizar para agrupar un conjunto de sentencias dentro de un ámbito concreto del programa. Un bloque de sentencias es un conjunto de instrucciones englobadas bajo llaves: '{' y '}'.

Hay diferentes lugares, en un programa escrito en C++, donde podemos usar bloques de código. Por ejemplo, en la sintaxis habitual de la función `main()`, todas las instrucciones que pertenecen a esta función principal se escriben dentro de una llave de apertura y una llave de cierre. Todo lo que hay entre esas dos llaves es el *código de la función principal*. De la misma manera, como se verá en el siguiente tema, cualquier otra función tiene sus llaves para agrupar las instrucciones que forman parte de ella.

Por otro lado, en C++ son posibles otros bloques de código que, como veremos en este tema, están asociados a estructuras de control y que engloban un conjunto de instrucciones que se ejecutan bajo las condiciones que controlan estas estructuras.

3.2. Operadores relacionales y lógicos

ANSI C++ define el tipo `bool` que tiene dos literales, `false` y `true`. Una expresión booleana o lógica es, por consiguiente, una secuencia de operandos y operadores que se combinan para producir uno de los valores `false` o `true`.

ANSI C no tiene tipos de datos lógicos o booleanos para representar los valores verdadero o falso. En su lugar utiliza el tipo `int` para ese propósito, con el valor 0 representando `false` y cualquier otro valor representando `verdadero`.

Operadores como `==` o `<=` que comprueban una relación entre dos operandos se llaman operadores relacionales y se utilizan en expresiones de la forma:

```
expresion1 operador_relacional expresion2
```

Los operadores relacionales se usan normalmente en sentencias de selección y de iteración para comprobar una condición. Operadores relacionales en C++:

Operador	Significado	Ejemplo
`==` (¡Ojo!, no =)	Igual a	`a == b`
`!=`	No igual a	`a != b`
`>`	Mayor que	`a > b`
`<`	Menor que	`a < b`
`>=`	Mayor o igual	`a >= b`
`<=`	Menor o igual	`a <= b`

Ejemplos:

```
x < 5.75    b * b >= 5.0 * a * c
```

Los operadores lógicos se utilizan con expresiones para devolver un valor verdadero o falso (`true` o `false`). Se denominan también operadores booleanos. Los operadores lógicos de C++ son: *not* (`!`), *and* (`&&`) y *or* (`||`). El operador *not* produce falso si su operando es verdadero, y viceversa. El operador *and* produce verdadero sólo si ambos operandos son verdaderos; si cualquiera de los operando es falso, produce falso. El operador *or* produce verdadero si cualquiera de los operandos es verdadero, y falso en caso de que los dos operandos sean falsos. Ejemplos:

```
!(7 == 5)   (aNum > 5) && (letra == 'a')    a >= 5 || b == 3 && c <= 8
```

3.3. Precedencia de operadores

La precedencia de operadores determina el orden en que se evalúan los operadores en una expresión. Seguiremos como referencia la siguiente lista, donde los operadores de cada grupo tiene prioridad sobre los del grupo siguiente:

1. `!` (not) `-` (menos unario) `+` (más unario)
2. operadores multiplicativos: `* / %`
3. operadores aditivos: `+ -` (binarios)
4. operadores relacionales de diferencia: `< <= > >=`
5. operadores relacionales de igualdad y desigualdad: `== !=`
6. operador lógico de conjunción: `&&`

7. operador lógico de disyunción: `||`

Además de esta lista, tenemos que tener en cuenta los siguientes puntos:

- Si dos operadores se aplican al mismo operando, el operador con más prioridad se aplica primero.
- Todos los operadores del mismo grupo tienen igual prioridad y asociatividad (se expresan de izquierda a derecha).
- Los paréntesis tienen la máxima prioridad.

3.4. Estructuras de selección

C++ tiene dos estructuras de control para la selección, `if` (selección simple y binaria) y `switch` (selección múltiple).

3.4.1. Estructura `if`

La sentencia `if` elige entre varias alternativas en base al valor de una o más expresiones booleanas.

La notación BNF de esta sentencia es la siguiente:

```
<sent_if> ::=   if (<expres_bool>) <bloque_sent>
                {else if (<expres_bool>) <bloque_sent>}
                [else <sec_sent>]
```

donde `<expres_bool>` es una expresión booleana y `<sec_sent>` puede ser una sola sentencia o un bloque de sentencias.

```
<sec_sent> ::=  ['{'{<sentencia>}'}'|<sentencia>]
```

Las formas más sencillas de esta sentencia, son :

```
if (<expres_bool>)
    <sentencia>
```

```
if (<expres_bool>)
    <sentencia>
else
    <sentencia>
```

```
if (<expres_bool>)
```

```
    if (<expres_bool>)
```

```
{
    <sent 1>
    ...
    <sent n>
}
```

```
{
  <sent 1>
  ...
  <sent n>
}
else
{
  <sent 1>
  ...
  <sent n>
}
```

Nótese que en el caso en el que sólo siga una sentencia al `if` no es necesario incluir las llaves, pero sí lo es cuando lo siguen más de una sentencia.

Los programas a menudo realizan una serie de tests de los que sólo uno será verdadero. Como ejemplo, considérese un programa para escribir en pantalla un mensaje diferente correspondiente a un número que representa una calificación numérica. En C++, utilizando sentencias `if` anidadas, quedaría:

```
if (Nota == 10)
   cout << "Matricula de Honor";
else
   if (Nota >= 9)
       cout << "Sobresaliente";
   else
       if (Nota >= 7)
           cout << "Notable";
       else
           if (Nota >= 5)
               cout << "Aprobado";
           else
               cout << "Suspenso";
```

Sin embargo, C++ nos proporciona una forma más concisa de expresar lo anterior, completamente equivalente:

```
if (Nota == 10)
    cout << "Matricula de Honor";
```

```
else if (Nota >= 9)
    cout << "Sobresaliente";
else if (Nota >= 7)
    cout << "Notable";
else if (Nota >= 5)
    cout << "Aprobado";
else
    cout << "Suspenso";
```

De esta última forma evitamos tener que usar numerosas sangrías del código que pueden dificultar la lectura. Por tanto, el criterio que se debe seguir para decidir el anidamiento o no de estructuras selectivas es la legibilidad del código.

3.4.2. Estructura `switch`

Aunque la sentencia `if` de C++ es muy potente, en ocasiones su escritura puede resultar tediosa, sobre todo en casos en los que el programa presenta varias elecciones después de chequear una expresión: selección múltiple o multialternativa. En situaciones donde el valor de una expresión determina qué sentencias serán ejecutadas es mejor utilizar una sentencia `switch` en lugar de una `if`. Por ejemplo, la sentencia `switch`:

```
switch (operador) {
    case + :
        resultado = A + B;
        break;
    case - :
        resultado = A - B;
        break;
    case * :
        resultado = A * B;
        break;
    case / :
        resultado = A / B;
        break;
    default :
        cout << "Operador invalido";
}
```

tiene el mismo efecto que la siguiente sentencia `if`:

```
if (operador == +)
    resultado = A + B;
else if (operador == -)
    resultado = A - B;
else if (operador == *)
    resultado = A * B;
else if (operador == /)
    resultado = A / B;
else
    cout << "Operador invalido";
```

En este ejemplo se observa que el valor de la expresión operador (en este caso sólo una variable) determina qué sentencias se van a ejecutar. Esta expresión recibe el nombre de selector de la estructura selectiva múltiple. También se puede apreciar la mayor simplicidad de la primera sentencia.

La palabra reservada `break` permite que el flujo de programa se detenga justo después de la ejecución de la sentencia anterior a ese `break`, impidiendo que se ejecuten las sentencias correspondientes a las siguientes alternativas del `switch`. Por tanto, debemos obligatoriamente acabar cada bloque de sentencias correspondiente a cada alternativa con una sentencia `break`.

Por otro lado, la alternativa `default` es opcional y engloba un conjunto de sentencias (que puede ser vacío, contener una sola sentencia o varias) que se ejecutan en caso de que ninguna de las alternativas del `switch` tenga un valor coincidente con el resultado de evaluar la expresión del selector. La notación BNF de la sentencia switch es:

```
<sent_switch>::= switch (<expresion>)
                 {
                     {case <caso> [break]}
                     [default <sec_sent>]
                 }
<caso> ::= <expresion_constante> : <sec_sent>
```

Una sentencia `switch` contiene un selector (en el ejemplo, operador), cuyo tipo debe ser `int`, `char` o enumerado. Cuando una sentencia `switch` se ejecuta, el valor del selector se compara con las etiquetas `case`. Si alguna de ellas concuerda con ese valor se ejecutará la correspondiente secuencia de sentencias. Si queremos que varias alternativas

tengan el mismo conjunto de sentencias a ejecutar, podemos hacer como en el siguiente ejemplo:

```
switch (selector)
{
     case 1:
     case 2:
          cout << "Salida para los casos 1 y 2" << endl;
          break;
     case 3:
          cout << "Salida para el caso 3" << endl;
          break;
     default:
          cout << "Salida para los restantes casos" << endl;
}
```

En este ejemplo, si el selector se evalúa y su valor es 1 ó 2, se ejecuta, en ambos casos, la instrucción `cout << "Salida para los casos 1 y 2"<< endl;`. En este caso particular puede apreciarse la utilidad de `break` a la hora de detener el flujo del programa.

La sentencia `switch` puede incluir la opción `default` para establecer la secuencia de sentencias a ejecutar en el caso de que ninguna etiqueta concuerde con el valor de la expresión `case`. El tipo de esta expresión `case` y el de las etiquetas tiene que ser el mismo.

3.5. Estructuras de iteración

C++ nos ofrece 3 esquemas de iteración diferentes:

- `while`
- `do while`
- `for`

3.5.1. Estructura `while`

La sintaxis de la sentencia `while` viene definida por la siguiente notación BNF:

```
<sent_while> ::= while (<expres_bool>)
                      <sec_sent>
```

Lo más importante a recordar de la sentencia while es que su condición de terminación (<expres_bool>) se comprueba cada vez antes de que el cuerpo del bucle (<sec_sent>) sea ejecutado. El cuerpo se ejecuta mientras se cumpla la condición de control. Si la condición es `false` (0), entonces el cuerpo no se ejecuta. Hay que hacer notar que, si la condición es `true` (1) inicialmente, la sentencia while no terminará (bucle infinito) a menos que en el cuerpo de la misma se modifique de alguna forma la condición de control del bucle. Una sentencia `while` se ejecutará cero o más veces.

Por ejemplo, si queremos leer una serie de enteros y encontrar su suma, parando cuando se lea un número negativo, podemos hacer:

```
sum = 0;
cin >> i;
while (i >= 0)
{
    sum = sum + i;
    cin >> i;
}
```

3.5.2. Estructura `do/while`

Su sintaxis viene dada por :

```
<sent_do_while> ::=    do
                           <sec_sent>
                       while (<expres_bool>)
```

Al igual que en la sentencia `while`, en esta sentencia el cuerpo del bucle se ejecuta mientras que sea verdad la expresión booleana que constituye la condición de control. Además, esta condición se comprueba cada vez tras la ejecución del cuerpo, no antes, como en el `while`. El cuerpo de una sentencia `do/while` siempre se ejecuta al menos una vez. Cuando esto pueda ocurrir es más conveniente su uso que la sentencia `while`. Por ejemplo, en lugar de escribir:

```
cout << "Introduzca un numero entre 1 y 10";
cin >> numero;
while (!((1 <= numero) && (numero <= 10))){
    cout << "Introduzca un numero entre 1 y 10" << endl;
    cin >> numero;
}
```

podemos hacer:

```
do {
    cout << "Introduzca un numero entre 1 y 10" << endl;
    cin >> numero;
}while (!((1<=numero) && (numero <= 10)));
```

La primera ejecución del bucle da a `numero` un valor, de manera que no necesitamos inicializarlo antes del bucle.

3.5.3. Estructura `for`

Su sintaxis:

```
<sent_for> ::= for (<expres_ini>;<expres_bool>;<expres_inc>)
                   <sec_sent>
```

El bucle for contiene las cuatro partes siguientes:

- Parte de inicialización (`<expres_ini>`), que inicializa las variables de control del bucle. Se puede utilizar variables de control de bucle simples o múltiples. Lo más normal es inicializar en este punto una sola variable cuyo valor varía luego en la parte de incremento. Si se inicializan varias variables de control, cada inicialización se separa de la anterior con una coma.

- Parte de iteración (`<expres_bool>`), que contiene una expresión lógica que hace que el bucle realice las iteraciones de las sentencias, mientras que a expresión sea verdadera.

- Parte de incremento (`<expres_inc>`), que modifica la variable o variables de control de bucle. Si se modifican varias variables de control, cada operación se separa de la anterior por una coma.

- Sentencias (¡sec_sent¿), acciones o sentencias que se ejecutarán por cada iteración del bucle.

Puede verse claramente la equivalencia entre una sentencia for y un while:

```
for (v = valor1; v<=valor2; v=v+paso)
{
    // sentencias
```

```
v = valor1;
while (v <= valor2)
{
```

```
    }                                                    // sentencias
                                                         v = v + paso;
                                                     }
```

Ejemplo de un bucle for con varias variables de control:

```
for (v1=valor1, v2=valor2; v1+v2<=100; v1++, v2++){
        // sentencias
}
```

Conviene tener en cuenta algunas consideraciones:

- Debemos asegurarnos que la expresión de inicialización del bucle y la expresión de incremento harán que la condición del bucle se convierta en falsa en algún momento.
- Si el cuerpo de un bucle (secuencia de sentencias) modifica los valores de cualquiera de las variables implicadas en ese bucle, entonces el número de repeticiones se puede modificar. Ejemplo:

  ```
  int limite = 1;
  int i;

  for (i=0; i<=limite; i++)
  {
      cout << i << endl;
      limite++;
  }
  ```

  produciría una secuencia infinita de enteros. No es una buena práctica de programación modificar el valor de la variable de control, por lo que evitaremos hacerlo.

3.6. Ejemplo

Para terminar esta segunda sección veamos un programa ejemplo en el que aparecen algunas de las estructuras vistas anteriormente. El programa encuentra el primer número perfecto mayor que 28, que es 496 (un número perfecto es aquel para el que la suma de todos sus divisores es igual al valor de dicho número).

```
/*-------------------------------------------------------------------*/
/* Autor:                                                            */
/*  Fecha:          Version: 1.0                                     */
/*-------------------------------------------------------------------*/
/* Programa Ejemplo para el calculo del primer numero perfecto       */
/*  mayor que 28                                                     */
/*-------------------------------------------------------------------*/
#include <iostream>
#include <cstdlib>

using namespace std;

int main()
{
    bool encontrado;
    unsigned int intento, cont, suma;

    encontrado = false;
    intento = 29;

    while (!encontrado) // Hasta que encontremos el numero
    {
        suma = 1;
        for (cont = 2; cont<intento; cont++)
        {
            if (!(intento%cont)) // Si es un divisor
            {
                suma = suma + cont;
            }
        }
        if (suma == intento)
        {
            encontrado = true;
        }
        else
        {
```

```
                intento = intento + 1;
            }
        }
        cout << "Numero perfecto mayor que 28 = " << intento << endl;

        return 0;
    }
```

Aunque existe otra forma más eficiente de resolver este problema, el programa anterior nos da una visión de como se pueden utilizar conjuntamente las distintas estructuras de control vistas en esta sección. Conviene hacer hincapié principalmente en que los anidamientos de estructuras tienen que ser correctos, es decir, que una estructura anidada dentro de otra lo debe estar totalmente.

Ejercicios

1. Hacer un programa que lea un número entero y escriba en pantalla su valor absoluto.

2. Una empresa maneja códigos numéricos con las siguientes características:

   - Cada código consta de cuatro dígitos,
   - El primero representa a una provincia,
   - El segundo el tipo de operación
   - Los dos últimos el número de la operación

3. Escriba un programa que lea de teclado un número de cuatro dígitos, y posteriormente imprima en pantalla la siguiente información.

   ```
   PROVINCIA              &
   TIPO DE OPERACION      &
   NUMERO DE OPERACION    &&
   ```

   En caso de que el número no tenga exactamente cuatro dígitos, en lugar del mensaje anterior, habrá que imprimir en pantalla el siguiente mensaje de error:

   ```
   ERROR: CODIGO INVALIDO.
   ```

4. Una compañía de gas calcula su factura trimestral tomando la lectura actual del contador y la del trimestre anterior y cobrando 35 céntimos de euro por unidad por las primeras 40 unidades y 25 céntimos de euro por unidad por el resto. Desarrolle un programa que reciba las dos lecturas del contador de un cliente y produzca una factura detallada.

5. El recibo de la electricidad se elabora de la siguiente forma:

   - 90 cntimos de euro de gastos fijos.
   - 50 cntimos/Kw para los primeros 100 Kw.
   - 35 cntimos/Kw para los siguientes 150 Kw.
   - 25 cntimos/Kw para el resto.

   Elabore un programa que lea de teclado los dos últimos valores del contador, y calcule e imprima en pantalla el importe total a pagar.

6. Escriba un programa que lea cuatro números enteros y escriba en pantalla el mayor de ellos. Sugerencia: acumule en una variable temporal uno de ellos y compare con los demás uno a uno.

7. Codifique un programa que se comporte como una calculadora simple. Para ello deberá tener las siguientes características:

   - Sólo efectuará operaciones con dos operandos.
   - Operaciones permitidas: (+,-,*,/).
   - Se trabajará con operandos enteros.
   - Pedirá en primer lugar el operados, y a continuación los dos operandos. Si el operador no se corresponde con alguno de los indicados se emitirá un mensaje de error.

   Ejemplo:

```
Operacion   : *
Operando 1  : 24
Operando 2  : 3
Resultado   : 72
Operacion   : *
```

8. Escriba un programa que resuelva completamente una ecuación de segundo grado, tanto si tiene raíces reales como imaginarias. Los coeficientes se leerán de teclado. Ejemplo:

```
Primer coeficiente  :   1
Segundo coeficiente :   0
Tercer coeficiente  :   1

SOLUCIONES : 0+i, 0-i
```

9. Escriba un programa que lea un número y escriba el día de la semana correspondiente o un mensaje de aviso si el número no corresponde a ningún día. Hágalo primero usando la sentencia `switch` y después usando sentencias `if`.

10. Escriba un programa que lea un número N de teclado y a continuación imprima en pantalla:

    - Su factorial.
    - La suma de los N primeros términos de la serie armónica (1+1/2+...+1/N)
    - Los n primeros términos de la sucesión de Fibonacci.

    Utilice una estructura repetitiva distinta para cada caso. ¿Cual es más apropiada?

11. Dado un número entero que representa una determinada cantidad de dinero, escriba un programa que lea dicho número del teclado y lo desglose para saber cuantos billetes de 500, 200, 100, 50, 20, 10 y 5 euros se necesitan.

12. Escriba un programa que lea un número N e imprima una pirámide de números con N filas como en la siguiente figura:

```
   1
  121
 12321
1234321
```

13. Calcule e imprima en pantalla los N primeros números primos, siendo N un número que se introduce por teclado. Posteriormente, modifique el programa para que muestre todos los números primos que hay entre 1 y N.

14. Calculador repetitivo. Modifique el programa **??** para que se repita un número indefinido de veces. El calculador dejará de trabajar cuando se introduzca como código de operación &. Ejemplo:

```
Operacion   : *
Operando 1  : 13
Operando 2  : 10
Resultado   : 130
Operacion   : u
ERROR!!!!
Operacion   : +
Operando 1  : 12
Operando 2  : 3
Resultado   : 15
Operacion   : &
FIN DEL PROGRAMA.
```

15. Escriba un programa que lea un número de teclado y después le dé al usuario la oportunidad de acertarlo en un número máximo de intentos. Supondremos que el usuario no sabe cuál es el número que se ha escrito.

16. Haga un programa que calcule la potencia de un número entero elevado a un número natural sólo en base a sumas. Sugerencia: recuerde que el producto se puede hacer en base a sumas.

17. Escribir un programa que lea una secuencia de letras y la escriba en pantalla codificada, sustituyendo cada letra por la letra que está tres posiciones después.

18. Escriba un programa que determine si la cadena `abc` aparece en una sucesión de caracteres cuyo final viene dado por un punto.