Lenguaje C++

Sitio:	Centros - Cádiz	Imprimido por:	Barroso López, Carlos
Curso:	Programación	Día:	martes, 21 de mayo de 2024, 23:25
Libro:	Lenguaje C++		

Tabla de contenidos

1. Introducción

1. El lenguaje C++

C++ es un lenguaje de programación **compilado**, **multiparadigma**, principalmente de tipo imperativo y orientado a objetos, incluyendo también programación genérica y funcional.

C++ es una **evolución** del **lenguaje C** el cual incluye mecanismos que permiten la manipulación de objetos (una forma de programar más avanzada). En ese sentido, desde el punto de vista de los lenguajes orientados a objetos, C++ es un ***lenguaje híbrido***.

2. Historia

Lenguaje C (predecesor de C++)

La historia de C parte de un lenguaje anterior, el lenguaje B, escrito por Ken Thompson en 1970 con el objetivo de recodificar el **sistema operativo UNIX**, que hasta el momento se había programado en ensamblador.

En 1972 es *Dennis Ritchie* (de los Laboratorios Bell de AT&T) quien diseña finalmente C a partir del B de Thompson, el cual permite realizar una **programación estructurada** economizando las expresiones (pocas palabras reservadas). Es un lenguaje considerado de **nivel medio**, debido a que proporcionada funcionalidades de alto y bajo nivel.

A principios de los 90 el estándar es reconocido por la ISO (Organización Internacional de Estándares) y comienza a comercializarse con el nombre **ANSI C**.

Lenguaje C++

El nombre **"C++"** (antes se había usado el nombre "C con clases") significa **"incremento de C"**, haciendo referencia al operador ++ (incremento) utilizado en C/C++.

En 1980 surge C++ de la mano de *Bjarne Stroustrup* (Laboratorios Bell de AT&T). Diseña este lenguaje con el objetivo de añadir a C nuevas características, como son:

- **Clases** y funciones virtuales. (de SIMULA 67)
- **Tipos genéricos** y expresiones. (de ADA)
- Posibilidad de **declarar variables en cualquier punto del programa**. (de ALGOL 68)
- Un auténtico motor de objetos con herencia múltiple que permite combinar la programación imperativa de C con la programación orientada a objetos.

Estas nuevas características mantienen siempre la esencia del lenguaje C: otorgan el control absoluto de la aplicación al programador, consiguiendo una **velocidad** muy superior a la ofrecida por otros lenguajes.

Otro hecho fundamental en la evolución de C++ es sin duda la incorporación de la **librería STL** años más tarde. Esta librería de clases con contenedores y algoritmos genéricos proporciona a C++ una potencia única entre los lenguajes de alto nivel.

En 1990 se reúnen las organizaciones ANSI e ISO para definir un **estándar** que formalice el lenguaje. El proceso culmina en 1998 con la aprobación del **ANSI C++**.

2. Estructura de un programa en C++

1. "Hola Mundo"

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

/* Función main()
Es la función donde empieza la ejecución */

int main() {
   cout << "Hola Mundo!!!"; // Imprime Hola Mundo
   return 0;
}
```

2. Elementos

Elementos que forman parte de la estructura de un programa en C++:

- `#include <iostream>` Instrucción para incluir la librería `iostream`. En esta librería están definidas las **funciones de entrada/salida (E/S)**, por ejemplo `cout`.
- `using namespace std;` Para utilizar el **espacio de nombres** estándar (`std`). Como podemos tener diferentes elementos en el lenguaje que se llamen igual, se utilizan espacios de nombres para agruparlas. Las funciones de E/S, como `cout` o `cin`, están definidas en el el espacio de nombres `std`, por lo tanto, con esta instrucción se evita tener que especificar dicho espacio de nombres cada vez que se utilicen estas funciones:

  ```
  std::cout << "Hola Mundo!!!";
  ```

- `int main()` Es la **función principal** del programa. Al ejecutar el programa son las instrucciones de esta función las que se empiezan a ejecutar. La función principal devuelve un valor entero (`int`) al sistema operativo (SO). Si el programa va a tener parámetros en la línea de comandos, la función main se define de la siguiente manera:

  ```
  int main(int argc, char *argv[])
  ```

- `cout << "Hola Mundo!!!";` Instrucción que imprime en pantalla.
- `return 0;` La función `main()` devuelve un valor entero al SO: **0 si todo ha salido bien**, distinto de 0 si se ha producido algún error. Esta instrucción devuelve el valor 0.
- Los bloques de instrucciones se guardan entre los caracteres `{` y `}`.
- Todas las instrucciones deben acabar en `;`.
- En C++ permite **comentarios de una línea** (utilizando los caracteres `//`) o comentarios de **varias líneas** (con los caracteres `/*` y `*/`). Todos los comentarios son ignorados por el compilador.
- El lenguaje C++ distingue entre mayúsculas y minúsculas. Hay ciertas convenciones al nombrar, por ejemplo, el nombre de las variables se suele poner siempre en minúsculas, mientras que el nombre de las constantes se suele poner en mayúsculas.

3. Tipos de datos

El tipo de dato representa la clase de datos con el que vamos a trabajar.

1. Clasificación

Podemos clasificar los tipos de datos de la siguiente manera:

- **Tipos de datos simples**:
  - Números enteros (`int`)
  - Números reales (`float` o `double`)
  - Valores lógicos (`bool`)
  - Caracteres (`char`)
- **Tipos de datos complejos**:
  - Arrays
  - Cadena de caracteres
  - Estructuras de datos

2. Representación

Los datos de un programa se pueden indicar de tres formas distintas:

2.1. Literales

Los literales permiten representar valores. Estos valores pueden ser de diferentes tipos, de esta manera tenemos diferentes tipos de literales:

- **Literales enteros**: Para representar números enteros. Ejemplos: números en base decimal (`5`,`-12`…), en base octal (`077`) y en hexadecimal (`0xfe`).
- **Literales reales**: Se utiliza un punto para separar la parte entera de la decimal. Por ejemplo: `3.14159`. También se puede usar la letra `e` o `E` seguida de un exponente con signo para indicar la potencia de 10 a utilizar, por ejemplo: `6.63e-34`, `35E20`.
- **Literales booleanos o lógicos**: Los valores lógicos solo tienen dos valores: `false` para indicar el valor falso, y `true` para indicar el valor verdadero.
- **Literales carácter**: Para indicar un valor de tipo carácter se utiliza la comilla simple `'`. Por ejemplo `'a'`. tenemos algunos **caracteres especiales** que son muy útiles, por ejemplo `\n` indica nueva línea y `\t` indica tabulador.
- **Literales cadenas de caracteres**: Una cadena de caracteres es un conjunto de caracteres. Para indicar cadenas de caracteres se utiliza la comilla doble `"`, por ejemplo: `"Hola"`.

2.2. Constantes

Una constante es un valor que identificamos con un nombre cuyo valor no cambia durante la ejecución del programa.

Existen **dos formas** de definir una constante:

i. Instrucción **#define**

```
#define identificador valor
```

Ejemplo:

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

#define ANCHURA 10
#define ALTURA  5
#define NUEVALINEA '\n'

int main() {
   int area;

   area = ANCHURA * ALTURA;
   cout << area;
   cout << NUEVALINEA;
   return 0;
}
```

La instrucción *#define* no almacena ningún valor constante en memoria, en realidad define una etiqueta que tiene asociada un valor. Esta etiqueta es sustituida por su valor durante la compilación.

ii. Modificador **const**

Utilizando el modificador `const` al crear una variable, se le indica al compilador que dicho valor no se va a modificar (modificación no permitida).

```
const float PI  = 3.1416; //Definimos una constante llamada PI
cout << "Mostrando el valor de PI: " << PI << endl;
```

2.3. Variables

Las variables son posiciones en memoria donde se almacena un dato de un determinado tipo.

Cada variable tiene un nombre (**identificador**) y al crearlas (declaración) hay que indicar el **tipo** de datos que va a almacenar.

C++ usa **tipado estático**, que caracteriza a aquellos lenguajes en los que el tipo asociado a una variable debe **fijarse antes de uso** en las diferentes partes del programa.

El tipado estático permite comprobar en **tiempo de compilación** la correcta asignación de cada variable a un determinado tipo y, en su caso, emitir los **avisos** y **errores** pertinentes.

Declaración

La declaración consiste en indicar el tipo y el nombre de la variable. Además, en una declaración también es posible **inicializar** la variable con un valor.

```
int variable1;
int variable2=10;
int variable3, variable4, variable5;
int variable6=100, variable7=-10;
```

Inicialización

Antes de utiliza una variable en una expresión es muy importante asegurarse de que está inicializada, es decir, que tiene un valor.

Dependiendo del lenguaje (y del compilador), las variables no inicializadas tendrán un determinado valor por defecto, o incluso no tener un valor definido (en C contienen "basura").

3.1. Operadores de asignación

1. Asignación simple

El operador `=` se utiliza para **modificar** el valor de las variables, es decir, altera el bloque de memoria asociado a una variable.

Ejemplo:


```
y = y + 1.5;
```

El resultado de la expresión `y + 1.5`, es asignado a la variable `y`, la cual quedará con el valor `3.5`.

- Parte derecha del operador (=**Rvalue**): valor asignado a la variable, puede ser un literal, otra variable, o el resultado de una expresión.

- Parte izquierda del operador (**Lvalue**=): nombre de la variable.

Ejemplos:

```
y = (2 + 3.5)/3.1 + a;  // OK
vel = x;                // OK
masa = 5.6;             // OK
x + y = u + w;          // ERROR
2 = z;                  // ERROR
```

```
x = a = 2;  // Primero se asigna a = 2, luego x = a
```

2. Asignación compuesta

Para una sentencia del tipo `a = a OP b;`, donde `OP` es uno de los **operadores aritméticos** (+, -, *, /, %), se puede utilizar la alternativa `a OP= b;`, más corta y eficiente.

```
a += b;    // equivale :  a = a + b
a -= b;    // equivale :  a = a - b
a *= b;    // equivale :  a = a * b
a /= b;    // equivale :  a = a / b
a %= b;    // equivale :  a = a % b
```

3. Incremento / Decremento

Operador **incremento** `++`

La sentencia a = a + 1; puede abreviarse por a++;

Operador **decremento** `--`

La sentencia a = a - 1; puede abreviarse por a--;

3.2. Tipos primitivos

Son los tipos de datos básicos o fundamentales que soporta el lenguaje.

La memoria asociada a un tipo fundamental en C++ se caracteriza por:

1. El **número de celdas** (bytes) es **fijo**.
2. Las celdas son **contiguas**.
3. Hay un **único valor** representado, no hay información accesoria ni metadatos.

1. Tipos primitivos de C++

En el caso de C++ son los siguientes:

Nombre	**Descripción**	**Tamaño (bytes)**	**Rango de valores**
bool	Booleano	1	true o false
char	Carácter o Entero pequeño	1	-128 a 127 o 0 a 255
int	Entero	4	-2.147.483.648 a 2.147.483.647
float	Decimal simple	4	Hasta 6 decimales
double	Decimal doble	8	Hasta 14 decimales

2. Modificadores de tipos

Permiten modificar el tamaño y el conjunto de valores que puede tomar un determinado tipo de dato.

2.1. **signed** (por omisión) y **unsigned**

Para representar valores **con signo** o **sin signo** respectivamente. El uso del modificador *unsigned* permite aumentar el valor máximo permitido

2.2. **short** y **long**

Para **disminuir** o **aumentar** el tamaño del tipo.

La siguiente tabla muestra las posibles combinaciones de modificadores de tipos:

Nombre	**Descripción**	**Tamaño (bytes)**	**Rango de valores**
`char`	Carácter o entero pequeño	1	con signo: -128 a 127 sin signo: 0 a 255
`short int` (`short`)	Entero corto	2	con signo: -32768 a 32767 sin signo: 0 a 65535
`int`	Entero	4	con signo: -2147483648 a 2147483647 sin signo: 0 a 4294967295
`long long int`	Entero largo doble	8	con signo: -9.223.372.775.808 a 9.223.375.775.807 sin signo: 0 a 2^64
`bool`	Valor booleano	1	true o false
`float`	Punto flotante	4	Hasta 6 decimales
`double`	Punto flotante de doble precisión	8	Hasta 14 decimales
`long double`	Punto flotante de doble precisión largo	12	Hasta 18 decimales

Ejemplo:

unsigned short int -> admite valores enteros entre 0 y 65535

3.3. Conversión de tipos (Casting)

El casting puede ser implícito (lo hace el lenguaje) o explícito (lo hace el programador).

1. Implícito

```
int i = 10;  // i vale 10 (entero)
float d = i; // d vale 10.0 (decimal)
```

- **Expresiones** con operaciones entre **datos del mismo tipo**: el resultado será del mismo tipo.
- Expresiones con operaciones entre datos de **distintos tipos**: el resultado será del tipo con más precisión de los datos operados.

2. Explícito

Se consigue escribiendo a la izquierda de una variable o expresión el tipo al que se quiere convertir:

```
int num1=10, num2=3;
float res;
res = (float)num1 / float(num2) //Dos formas de hacer la conversión
```

Ejemplos

Ejemplo 1:

```
int num1=4, num2=3;
cout << 4 / 3 << endl; (Salida= 1)
cout << 4.0 / 3 << endl; (Salida= 1.333)
cout << float(num1) / num2 << endl; (Salida= 1.333)
```

Ejemplo 2:

```
int a = 10;
int b = 7;
float c = a / b;
```

¿Cuánto vale c?

4. Entrada y Salida (E/S)

1. Biblioteca iostream

La forma más elemental de proporcionar datos es vía **teclado** y, de mostrarlos, es utilizando una **terminal** o consola, interfaz gráfica que muestra únicamente caracteres alfanuméricos.

C++ proporciona estos recursos básicos a través de la **biblioteca de entrada/salida** `iostream`, que forma parte de la *biblioteca estándar de C++*.

Para hacer uso de esta biblioteca en un programa es necesario incluirla mediante la siguiente línea al inicio del código:

```
#include <iostream>
```

2. espacio de nombres "std"

Todas las funcionalidades de la **biblioteca estándar** y, por tanto, las de `iostream`, se encuentran englobadas dentro de un **espacio de nombres** (**namespace**) denominado `std`.

Los espacios de nombres proporcionan un ámbito único a un grupo de identificadores, **evitando conflictos de nombres** que pueden producirse cuando el programa utiliza varias bibliotecas.

Para poder usar cualquiera de los identificadores perteneciente a un espacio de nombres debemos utilizar el espacio de nombres seguido del **operador de resolución de ámbito** `::`.

```
std::cout
```

La línea `using namespace std;` *informa* al compilador de que, si algún identificador no está declarado, entonces debe buscarlo en el espacio de nombres `std`.

```
#include <iostream>

using namespace std;

int main()
{
   cout << "Hola mundo";
}
```

4.1. cout

cout permite enviar datos a la consola para ser imprimidos como texto.

```
#include <iostream>
int main()
{
   std::cout << "Hola mundo.";
}
```

`<<` **operador de inserción**: permite insertar caracteres al flujo de salida `cout`

Salidas con variables

```
   int x = 3;
   double y = 8.13;
   cout << "x=" << x << ", y=" << y;
```

Múltiples líneas

```
   cout << "x=" << x;
   cout << "y=" << y;  // muestra "x=3y=8.13" en una sólo línea
```

Para incluir una **nueva línea** en la salida, puede utilizarse alguna de las siguientes alternativas:

- Incluir el carácter nueva línea: '**\n**'

- Utilizar `std::endl` de la biblioteca estándar.

Ejemplo:

```
   cout << "x=" << x << endl;
   cout << "y=" << y << "\n";
```

4.2. cin

cin permite leer información introducida por teclado y almacenarla en una variable.

Ejemplo:

```
cout << "Introduce un entero: ";
int x;
cin >> x;
cout << "x=" << x << '\n';
```

`>>` **operador de extracción**: permite extraer caracteres del flujo de entrada `cin` y transformarlos a su representación binaria interna.

Introducción de varios valores

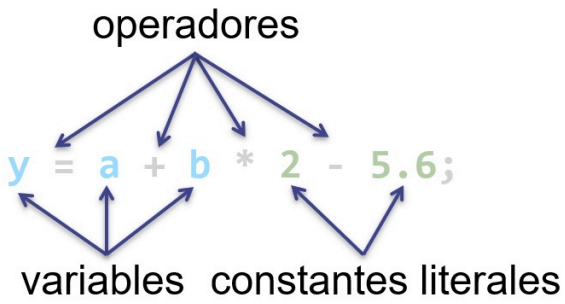
```
cout << "Introduce dos enteros: ";
int x, y;
cin >> x >> y;
cout << "x=" << x << "\ny=" << y << '\n';
```

La introducción de caracteres desde el teclado por parte del usuario es **libre** y, por tanto, sujeta a **errores**.

5. Expresiones y sentencias

1. Expresión

Las expresiones permiten combinar **datos** y **operadores** para calcular otros datos, es decir, obtener un resultado.



2. Sentencia

Una sentencia es un conjunto de expresiones que permiten **ejecutar** una determinada acción.

En C++, las **sentencias simples** se caracterizan por terminar con el signo de puntuación `;`.

```
(x + 3)*2          // expresión
a > b && c < d     // expresión
int x;             // sentencia
y = (x + 3)*2;     // sentencia
```

5.1. Operadores aritméticos

Permiten realizar operaciones aritméticas sobre sus operandos.

Op. aritméticos

Operador	Descripción
+	suma
-	resta
*	producto
/	división
%	módulo (resto de la división entera)

El operador `%` sólo es aplicable a datos enteros. Por ejemplo: `23%5` es `3`. Equivale a `23-(23/5)*5`

El **tipo** del valor generado dependerá de los tipos de los operandos.

Reglas de precedencia

Cuando se combinan diferentes operadores, el valor determinado por una expresión se calcula **siguiendo un orden de evaluación** en función de unas **reglas de precedencia**.

Precedencia de los operadores

Precedencia	Operador	Descripción	Asociatividad
1	`::`	Resolución de ámbito	Izquierda a derecha
2	`(expresion)`	Expresión entre paréntesis	Izquierda a derecha
5	`* / %`	Multiplicación, división y división entera	Izquierda a derecha
6	`+ -`	Suma y resta	Izquierda a derecha
7	`<< >>`	Extracción e inserción	Izquierda a derecha
16	`=`	Asignación	Derecha a izquierda

Ejemplos:

```
// Operaciones con constantes literales enteras
(3 + 2)*5         // 1º) (3 + 2) 2º) 5*5: Resultado 25
(3 + 2)*12/5      // 1º) (3 + 2), 2º) 5*12, 3º) 60/5: Resultado 12
(3 + 2)*(12/5)    // 1º) (3 + 2), 2º) (12/5) 3º) 5*2: Resultado 10
```

5.2. Operadores relacionales y lógicos

1. Tipo de dato booleano o lógico

El tipo booleano o lógico `bool` admite sólo dos posibles valores, expresados por los literales predefinidos: `true` (1) y `false` (0).

En realidad, cualquier valor entero distinto de 0 será verdadero, y el 0 será falso.

2. Operadores relacionales o de comparación

Sirven para **comparar** dos variables o expresiones. La **expresión boolena** resultante en la que intervienen puede resultar **cierta** (**true**) o **falsa** (**false**).

Op. relacionales

Operador	Descripción
>	mayor que
>=	mayor o igual que
<	menor que
<=	menor o igual que
==	igual que
!=	distinto que

C++ utiliza el tipo de

Las expresiones de comparación devuelven como resultado un valor lógico.

```
int x = 3;
int y = 4;
bool z = x > y;  // z tomará el valor false
cout << z << endl;
z = x != y;      // z tomará el valor true
cout << z << endl;
```

3. Operadores lógicos

Sirven para concatenar expresiones de tipo lógico, cuyo resultado, será también un valor lógico (`true` o `false`).

La evaluación de las operaciones lógicas se realiza de **izquierda a derecha** y se **interrumpe** cuando se ha asegurado el resultado.

Op. lógicos

Operador	Descripción
&&	**Y**, devuelve `true` cuando todas las expresiones que relaciona son ciertas
||	**O**, devuelve `true` cuando al menos una expresión es cierta
!	**NO**, devuelve `true` cuando la expresión a la que afecta es `false`

Las **tablas de verdad** asociadas a estos operadores son:

Operador **Y** lógico

```
a     b     a && b

false false false

false true  false

true  false false

true  true  true
```

Operador **O** lógico

```
a     b     a || b

false false false

false true  true

true  false true

true  true  true
```

Operador **NO** lógico

```
a          !a

false      true

true       false
```

4. Orden de precedencia

Tabla de precedencia de los operadores de comparación y lógicos

Precedencia	Operador	Descripción	Asociatividad
3	`!`	Negación lógica	Izquierda a derecha
9	`< <= > >=`	Operadores de comparación	Izquierda a derecha
10	`== !=`	Operadores de igualdad	Izquierda a derecha
14	`&&`	Conjunción lógica	Izquierda a derecha
15	`||`	Disyunción lógica	Izquierda a derecha

¡Importante!: Usar los paréntesis `(...)` en caso de duda y/o para mejorar la legibilidad.

5. Ejemplos

Ejemplo 1

```
// Ejemplo que controla el rango permitido de valores de dos enteros
#include <iostream>
using namespace std;

int main()
{
   int x, y;
   cout << "Introduce dos números enteros positivos o 0. "
        << "Ambos no pueden valer simultáneamente 0.\n";
   cout << "Introduce el primer número: ";
   cin >> x;
   cout << "Introduce el segundo número: ";
   cin >> y;

   if (x < 0 || y < 0)
      cout << "Error. Los números introducidos no son ambos positivos.\n";
   else if (x == 0 && y == 0)
      cout << "Error. Los dos números son 0.\n";
   else
      cout << "Los números introducidos son validos.\n";
}
```

Ejemplo 2

```
// Ejemplo que controla que un denominador no sea 0
#include <iostream>
using namespace std;

int main()
{
   int x, y;
   cout << "Introduce el numerador: ";
   cin >> x;
   cout << "Introduce el denominador: ";
   cin >> y;

   if (!y)  // Equivalente a if (y == 0)
      cout << "Error. No podemos dividir por 0.\n";
   else
      cout << "x/y= " << x/y << endl;  // ¡Ojo! división entera
}
```

5.3. Operadores incremento y decremento

Son operadores **unarios** (actúan sobre un único operando), incrementando o decrementado una variable entera en una unidad:

Op. incremento y decremento

Operador	Descripción
`++x`	**preincremento**: utiliza el valor ya incrementado en la sentencia
`x++`	**postincremento**: utiliza el valor antes de incrementar
`--x`	**predecremento**: utiliza el valor ya decrementado en la sentencia
`x--`	**postdecremento**: utiliza el valor antes de decrementar

Ejemplos:

```
++x;  // Ambos equivalen a "x = x + 1", aunque de forma MÁS EFICIENTE
x++;
```

```
int x = 10;
int y = ++x;  // x=11, y=??
```

```
int x = 10;
int y = x++;  // x=11, y=??
```

Aclaraciones:

- Los compiladores modernos pueden optimizar el código y transformar la sentencia `x = x + 1;` a, por ejemplo, `++x;` de forma transparente al usuario.

- Es más eficiente `++x` dado que con `x++` se tiene que guardar el valor incrementado de la variable `x`, que es un valor por la derecha, hasta la siguiente sentencia.

6. Estructuras alternativas

Una **sentencia condicional** realiza un conjunto u otro de sentencias dependiendo del cumplimiento o no de una determinada condición.

En C++ podemos distinguir diferentes tipos:

- **Simple**: `if`
- **Ampliada**: `if - else`
- **Anidada**: `if – else if - … - else`
- **Multisalida**: `switch`

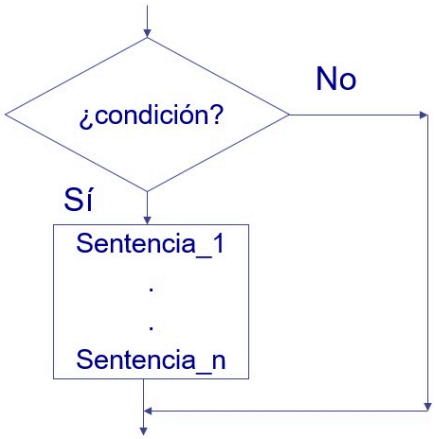
6.1. if (else)

Es una sentencia en la que se **evalúa una condición** y, solo si es verdad, se ejecuta un conjunto de sentencias asociadas.

1. Simple

```
if (condicion)
{
   bloque de sentencias;
}
```

Las llaves `{ … }` delimitan el conjunto de sentencias afectadas por la condición `if`.



Cuando el bloque está formado por una **única sentencia**, pueden omitirse las llaves.

```
if (a != 0)
{
   a = 1/a;
}
```

es equivalente a:

```
if (a != 0)
   a = 1/a;
```

El ejemplo anterior es habitual escribirlo como:

```
if (a)  // a != 0 <-> a
   a = 1/a;
```

dado que cualquier valor no nulo será siempre evaluado como cierto.

¡¡Advertencia!!

El siguiente fragmento ilustra un error habitual:

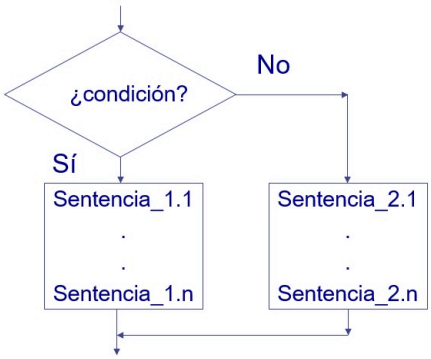
```
a = 5;
if (a = 1)   //¡¡ERROR!!
   cout << "La variable vale 1\n";
```

¡Se esta **asignando** en lugar de **comparando**!

2. Doble

Permite ejecutar dos bloques de sentencias distintas dependiendo de que se cumpla la condición o no.

```
if (condicion)
{
   bloque de sentencias 1;
}
else
{
   bloque de sentencias 2;
}
```



Ejemplo:

```
        int edad;
        cout << "Dime tu edad:";
        cin >> edad;
        if (edad>=18)
        {
                cout << "Eres mayor de edad" << endl;
        }
        else
        {
                cout << "Eres menor de edad" << endl;
        }
```

Errores habituales:

```
if (a != 0)
{
   b = a;
   a = 1/a;
};  // ; erróneo
else
   cout << "No se puede invertir el 0\n";
```

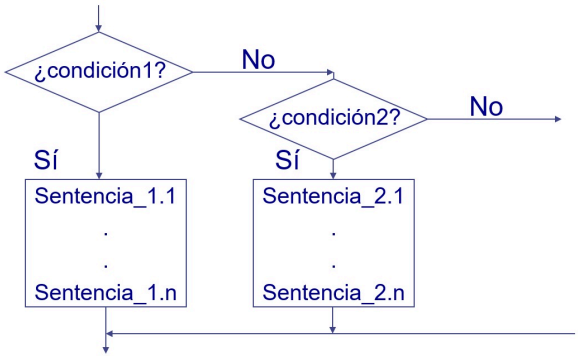
```
if (a != 0)  // Faltan llaves
   b = a;
   a = 1/a;
else
   cout << "No se puede invertir el 0\n";
```

```
if (a != 0);  // Error semántico
{
   b = a;
   a = 1/a;
}
```

6.2. if - else if - ... - else

La estructura *if - else if - else* permite evaluar de forma **anidada** varias expresiones condicionales de arriba a abajo. Cuando aparece una condición verdadera, ejecuta las acciones asociadas y salta el resto de las expresiones condicionales sin necesidad de evaluarlas.

```
if (condicion_1)
{
   bloque 1 de sentencias;
}
else if (condicion_2)
{
   bloque 2 de sentencias;
}
else if (condicion_3)
{
   bloque 3 de sentencias;
}
// Tantos bloques else if como sean necesarios
else // si no se cumple ninguna de las anteriores
{
   bloque n de sentencias;
}
```



Esta sintaxis no es más que una forma abreviada de representar estructuras if-else anidadas:

```
if (condicion_1)
{
   bloque 1 de sentencias;
}
else
{
   if (condicion_2)
   {
      bloque 2 de sentencias;
   }
   else
   {
      if (condicion_3)
      {
         bloque 3 de sentencias;
      }
      else // si no se cumple ninguna de las anteriores
      {
         bloque 4 de sentencias;
      }
   }
}
```

Importancia de una correcta tabulación o sangrado

Ilegible:

```
if (n > 0)
if (a > b)
z = a;
else
z = b;
```

Legible:

```
if (n > 0) {
   if (a > b)
      z = a;
   else
      z = b;
```

```
}
```

6.3. switch

La sentencia switch ejecuta un bloque de sentencias si una **variable o expresión entera** coincide con alguno de los valores proporcionados en una lista de **constantes enteras**.

```
switch (expresion)
{
   case valor_1:
      sentencias_1
   case valor_2:
      sentencias_2
   //Resto de bloques case
   case valor_n:
      sentencias_n
   default:
      sentencias_default;
}
```

La ejecución de esta estructura consiste en comparar `expresion` de arriba a abajo con cada constante literal que aparece tras la palabra reservada `case`. Si se localiza una coincidencia, se ejecutan **todas** las sentencias por debajo de esa constante hasta:

- finalizar el bloque `switch`
- encontrar una sentencia `break;`

Opcionalmente se puede insertar una condición por defecto mediante `default`, cuyas sentencias asociadas se ejecutarían si no se ha encontrada previamente una coincidencia.

Ejemplo:

```
   cout << "Dame el mes: (1, 2, ..., 12): ";
   int mes;
   cin >> mes;

   if (mes > 0 && mes < 13)
      switch(mes)
      {
         case 1:
         case 3:
         case 5:
         case 7:
         case 8:
         case 10:
         case 12:
            cout << "El mes tiene 31 dias.\n";
            break;
         case 4:
         case 6:
         case 9:
         case 11:
            cout << "El mes tiene 30 dias.\n";
            break;
         case 2:
            cout << "El mes tiene 28 o 29 dias.\n";
            break;
         default:
            cout << "¡Imposible!\n";
      }
   else
      cout << "El valor introducido no es válido.\n";
```

7. Estructuras repetitivas o iterativas

Las sentencias **repetitivas** o **iterativas** permiten repetir una secuencia de instrucciones en tanto no deje de cumplirse una **condición**. Estas estructuras se denominan también **bucles** (**loops**).

En el momento que la **condición** pasa a ser falsa, la ejecución del programa continúa en la línea que sigue al bucle.

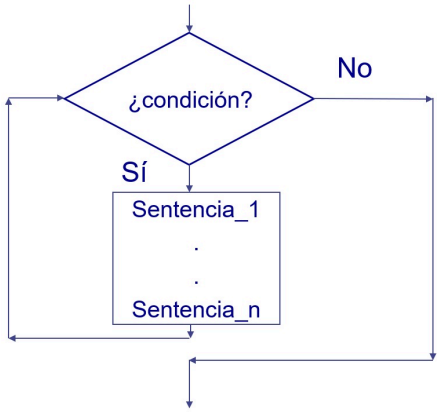
En C++ disponemos de tres variantes de bucles:

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

7.1. while

Un bucle `while` ejecuta un bloque de sentencias **mientras** la condición de entrada al bucle sea cierta.

```
while (condicion)
{
   bloque de sentencias;
}
```



La **condición se evalúa al comienzo** de la estructura. Esto supone que el bloque de instrucciones puede no ejecutarse ninguna vez si la condición es inicialmente falsa.

0 a N ejecuciones

Si la condición siempre es verdadera, al ejecutar esta instrucción se produce un **ciclo infinito**. A fin de evitarlo, el cuerpo del ciclo debe contener alguna instrucción que modifique la o las variables involucradas en la condición, de modo que ésta sea falsa en algún momento y así finalice la ejecución del ciclo.

Ejemplo:

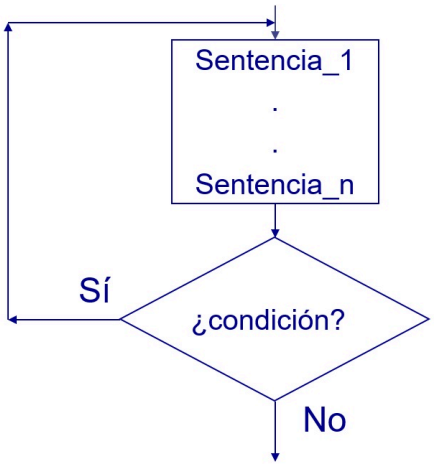
```
cout << "Dame el mes: (1, 2, ..., 12): ";
int mes;
cin >> mes;
while (mes < 1 || mes > 12)
{
   cout << "El valor introducido no es válido.\n";
   cout << "Dame el mes: (1, 2, ..., 12): ";
   cin >> mes;
}
```

7.2. do - while

Al contrario que el bucle `while`, el bucle `do - while` evalúa la condición al final del bucle. Esto implica que el bucle **se ejecutará al menos una vez**.

1 a N ejecuciones

```
do
{
   bloque de sentencias;
}
while (condicion);
```



El uso de `do - while` es muy habitual en los **menús** de introducción de datos por parte de un usuario.

1. Al menos una vez el usuario tendrá que introducir los datos
2. Si se detecta un error, se vuelven a solicitar

Ejemplo:

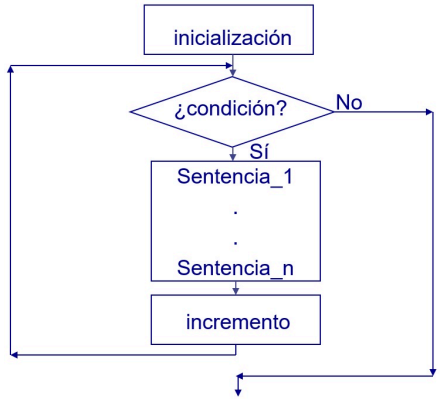
```
int mes;
do
{
   cout << "Dame el mes: (1, 2, ..., 12): ";
   cin >> mes;
   if (mes < 1 || mes > 12)
      cout << "El valor introducido no es válido.\n";
}
while (mes < 1 || mes > 12);
```

7.3. for

El bucle `for` está concebido fundamentalmente para ejecutar sus sentencias asociadas **un número fijo de veces**.

Por tanto, este bucle es adecuado cuando se conoce de antemano el número de iteraciones a realizar.

```
for (<variable> = <valor_inicial>; <condición>; <incremento/decremento variable>)
{
        <instrucciones>
}
```

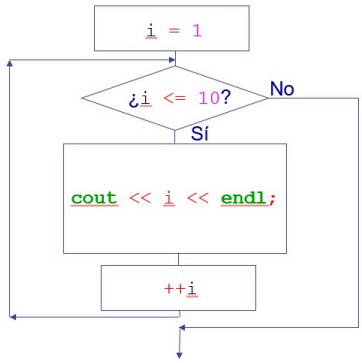


Como en el bucle `while`, en el bucle `for` se comprueba la **condición al inicio del bucle**, lo que significa que el código asociado puede que no se ejecute ni una sola vez.

Ejemplo:

```
int i;
for (i = 1; i <= 10; ++i)
{
   cout << i << endl;
}
```

Imprime por pantalla los números del `1` al `10`.



Declaración de la variable contador dentro del bucle

```
for (int i = 1; i <= 10; ++i)
{
   cout << i << endl;
}
```

Es una buena práctica porque limita el **ámbito** de la variable al bucle.

7.4. Elección del bucle apropiado

¿Se conoce el número de iteraciones a realizar de antemano?

- **SI**: la opción a elegir es usar un bucle `for`
- **NO**: entonces debemos hacernos una nueva pregunta:
  - ¿Se ejecutará al menos una vez el bloque de sentencias asociado al bucle?
    - **SI**: deberemos utilizar un bucle `do - while`
    - **NO**: el bucle `while` debería ser la elección

Las estructuras `for`, `do - while` y `while` se pueden transformar entre sí de forma relativamente sencilla.

```
for (int i = 0; i < 10; ++i)
{
   sentencias;
}

// equivale a

int i = 0;
while (i < 10)
{
   sentencias;
   ++i;
}
```

7.5. Variables de uso específico

1. Contador

Variable entera encargada de **contar** cuantas veces se ejecuta un bucle o ocurre en suceso.

- Inicialización: (antes del bucle)

```
cont = 0;
```

- Incremento (o decremento) en cada iteración del bucle o cuando ocurre el suceso a contar: (dentro del bucle)

```
cont = cont + 1;  // o cont++; (más eficiente)
```

En muchas ocasiones, la condición de salida de un bucle viene determinada por un contador.

2. Acumulador

Variable que permite ir acumulando el resultado parcial de una operación (habitualmente la suma) en cada iteración de un bucle. Al finalizar el bucle, el acumulador contiene el resultado final.

- Inicialización: (antes del bucle)

```
acum = 0;
```

- Acumula un valor intermedio: (dentro del bucle)

```
acum = acum + num;  // o acum += num; (más eficiente)
```

Ejemplo:

```
  cout << "Introduzca cuantos números reales quiere sumar: ";
  int num_valores;
  cin >> num_valores;

  int contador = 0;
  double suma = 0.0;
  while (contador < num_valores)
  {
     cout << "Deme valor real a sumar: ";
     double valor;
     cin >> valor;
     suma += valor;  // Equivalente a suma = suma + valor
     contador++;  // Equivalente a contador = contador + 1
  }
```

3. Bandera o "flag"

Variable lógica (true/false) utilizada para recordar o indicar algún suceso.

- Inicialización a un valor lógico que indica que el suceso no ha ocurrido:

```
indicador = false;
```

- Activación cuando ocurre el suceso:

```
indicador = true;
```

8. Saltos incondicionales

Los saltos incondicionales son sentencias que permiten **interrumpir** la secuencia natural de ejecución.

C++ dispone de 4 tipos de saltos incondicionales:

- `break`: Termina de forma inmediata la ejecución de un bucle (`for`, `while`, `do while`) o de una sentencia `switch`.
- `continue`: Finaliza la iteración actual de un bucle pasando a la siguiente.
- `goto`: Salta a una línea cualquiera (etiquetada) del programa.
- `return`: Salida de una función, pudiendo devolver un valor. Cualquier instrucción que haya posterior al return no será ejecutada.

Recomendaciones ("obligaciones") de uso

- Las instrucciones *continue* y *goto* rompen el flujo normal de ejecución de un programa, y **NO deben ser utilizadas**.

- La instrucción *break* se utiliza en las estructuras switch, debiéndose **evitar su uso en bucles**, dado que es posible escribir un código alternativo con la misma funcionalidad que no los utilice.

Ejemplo:

Con break:

```
int suma = 0;
for (int i = 0; i < 10; ++i)
{
  cout << "Introduzca un dato (negativo para finalizar): ";
  int dato;
  cin >> dato;
  if (dato < 0)
    break;
  suma += dato;
}
```

Sin break:

??

9. Números pseudoaleatorios

Para generar números pseudoaleatorios, el ordenador usará una **semilla** (seed) y le aplicará transformaciones matemáticas, convirtiéndola en otro número. Este nuevo número se convierte en la próxima semilla para el generador de números aleatorios.

Mientras el programa seleccione una semilla diferente en cada ejecución, (a efectos prácticos) nunca obtendrá la misma secuencia de números aleatorios. Lo que significa que si se utiliza la misma semilla, la secuencia aleatoria se repetirá.

Las transformaciones matemáticas utilizadas se seleccionan cuidadosamente para que todos los números generados aparezcan con la misma frecuencia y no muestren patrones obvios.

```
#include <iostream>
#include <ctime>    // biblioteca de tiempo de C

int main ()
{
    std::srand( std::time(nullptr));  // se establece como semilla la hora actual (en segundos)
    std::cout << std::rand() << std::endl;  // se genera un número aleatorio
}
```

La función `time` devuelve la cantidad de segundos desde el 1 de enero de 1970. Esta convención proviene del sistema operativo Unix y, a veces, se denomina **tiempo Unix**.

Dentro de un rango

Para conseguir números aleatorios dentro de un rango, se puede aplicar el siguiente algoritmo:

```
srand( time(nullptr) );  // se establece la semilla

int min = 0, max = 10;  // se define un mínimo y un máximo
int num = (rand() % (max - min + 1)) + min;  // se aplica esta operación
cout << num << endl;
```

Establecer la misma semilla generará siempre el mismo resultado.

Reiniciar tour para usuario en esta página