



Fundamentos de Programación

Curso 2019/2020

© Copyright: Juan Carlos Cubero. Universidad de Granada.

Sugerencias: por favor, enviar un e-mail a [JC.](mailto:JC.Cubero@decsai.ugr.es)

Cubero@decsai.ugr.es



El color marrón se utilizará para los títulos de las secciones, apartados, etc

El color azul se usará para los términos cuya definición aparece por primera vez. En primer lugar aparecerá el término en español y entre paréntesis la traducción al inglés.

El color rojo se usará para destacar partes especialmente importantes

Algunos símbolos usados:



Principio de Programación.



denota algo especialmente importante.



denota código bien diseñado que nos ha de servir de modelo en otras construcciones.



denota código o prácticas de programación que pueden producir errores lógicos graves.



denota una norma o consejo de programación especialmente importante, cuyo incumplimiento acarrea graves consecuencias en la evaluación de la asignatura.



denota código que nos da escalofríos de sólo verlo.



denota código que se está desarrollando y por tanto tiene problemas de diseño.



denota un consejo de programación.



denota contenido de ampliación. No entra como materia en el examen.



Reseña histórica.



denota contenido que el alumno debe estudiar por su cuenta. Entra como materia en el examen.

Contenidos

I. Introducción a la Programación	1
I.1. El ordenador, algoritmos y programas	2
I.1.1. El Ordenador: Conceptos Básicos	2
I.1.2. Datos y Algoritmos	3
I.1.3. Lenguajes de programación	6
I.1.4. Compilación	13
I.2. Especificación de programas	14
I.2.1. Organización de un programa	14
I.2.2. Elementos básicos de un lenguaje de programación	19
I.2.2.1. Tokens y reglas sintácticas	19
I.2.2.2. Palabras reservadas	20
I.2.3. Tipos de errores en la programación	21
I.2.4. Cuidando la presentación	23
I.2.4.1. Escritura de código fuente	23
I.2.4.2. Etiquetado de las Entradas/Salidas	24

I.3. Datos y tipos de datos	25
I.3.1. Representación en memoria de datos e instrucciones	25
I.3.2. Datos y tipos de datos	26
I.3.2.1. Declaración de datos	26
I.3.2.2. Inicialización de los datos	31
I.3.2.3. Literales	33
I.3.2.4. Datos constantes	33
I.3.2.5. Ámbito de un dato	39
I.4. Operadores y expresiones	40
I.4.1. Expresiones	40
I.4.2. Terminología en Matemáticas	42
I.4.3. Operadores en Programación	43
I.5. Tipos de datos simples en C++	45
I.5.1. Los tipos de datos enteros	46
I.5.1.1. Representación de los enteros	46
I.5.1.2. Rango de los enteros	47
I.5.1.3. Literales enteros	48
I.5.1.4. Operadores	49
I.5.1.5. Expresiones enteras	52
I.5.2. Los tipos de datos reales	54
I.5.2.1. Literales reales	54

I.5.2.2.	Representación de los reales	55
I.5.2.3.	Rango y Precisión	58
I.5.2.4.	Indeterminación e Infinito	60
I.5.2.5.	Operadores	61
I.5.2.6.	Funciones estándar	62
I.5.2.7.	Expresiones reales	63
I.5.3.	Operando con tipos numéricos distintos	64
I.5.3.1.	Asignaciones entre datos de distinto tipo .	64
I.5.3.2.	Asignaciones a datos de expresiones del mismo tipo	69
I.5.3.3.	Expresiones con datos numéricos de distinto tipo	72
I.5.3.4.	El operador de casting (Ampliación)	78
I.5.4.	El tipo de dato cadena de caracteres	80
I.5.5.	El tipo de dato carácter	85
I.5.5.1.	Representación de caracteres en el ordenador	85
I.5.5.2.	Literales de carácter	89
I.5.5.3.	Asignación de literales de carácter	91
I.5.5.4.	El tipo de dato <code>char</code>	93
I.5.5.5.	Funciones estándar y operadores	97
I.5.6.	Lectura de varios datos	98
I.5.7.	El tipo de dato lógico o booleano	103

I.5.7.1. Rango	103
I.5.7.2. Funciones estándar y operadores lógicos	103
I.5.7.3. Operadores Relacionales	106
I.6. El principio de una única vez	109
II. Estructuras de control	118
II.1. Estructura condicional	119
II.1.1. Flujo de control	119
II.1.2. Estructura condicional simple	122
II.1.2.1. Formato	122
II.1.2.2. Diagrama de flujo	124
II.1.2.3. Variables no asignadas en los condicionales	129
II.1.2.4. Cuestión de estilo	131
II.1.2.5. Estructuras condicionales consecutivas	132
II.1.2.6. Condiciones compuestas	135
II.1.3. Estructura condicional doble	137
II.1.3.1. Formato	137
II.1.3.2. Condiciones mutuamente excluyentes	143
II.1.3.3. Álgebra de Boole	148
II.1.3.4. Estructuras condicionales dobles consecutivas	152
II.1.4. Anidamiento de estructuras condicionales	158

II.1.4.1.	Funcionamiento del anidamiento	158
II.1.4.2.	Anidar o no anidar: he ahí el dilema	164
II.1.5.	Estructura condicional múltiple	172
II.1.6.	Ámbito de un dato (revisión)	175
II.1.7.	Algunas cuestiones sobre condicionales	177
II.1.7.1.	Cuidado con la comparación entre reales . .	177
II.1.7.2.	Evaluación en ciclo corto y en ciclo largo . .	179
II.1.8.	Programando como profesionales	180
II.1.8.1.	Diseño de algoritmos fácilmente extensibles	180
II.1.8.2.	Descripción de un algoritmo	190
II.1.8.3.	Descomposición de una solución en tareas	194
II.1.8.4.	Las expresiones lógicas y el principio de una única vez	199
II.1.8.5.	Separación de entradas/salidas y cálculos	201
II.1.8.6.	El tipo enumerado y los condicionales . . .	207
II.2.	Estructuras repetitivas	214
II.2.1.	Bucles controlados por condición: pre-test y post-test	214
II.2.1.1.	Formato	214
II.2.1.2.	Algunos usos de los bucles	216
II.2.1.3.	Bucles para lectura de datos	225
II.2.1.4.	Bucles sin fin	236
II.2.1.5.	Condiciones compuestas	238

II.2.1.6. Bucles que buscan	242
II.2.2. Programando como profesionales	245
II.2.2.1. Evaluación de expresiones dentro y fuera del bucle	245
II.2.2.2. Bucles que no terminan todas sus tareas . .	248
II.2.2.3. Estilo de codificación	252
II.2.3. Bucles controlador por contador	253
II.2.3.1. Motivación	253
II.2.3.2. Formato	255
II.2.4. Ámbito de un dato (revisión)	262
II.2.5. Anidamiento de bucles	265
II.3. Particularidades de C++	275
II.3.1. Expresiones y sentencias son similares	275
II.3.1.1. El tipo <code>bool</code> como un tipo entero	275
II.3.1.2. El operador de asignación en expresiones .	277
II.3.1.3. El operador de igualdad en sentencias . . .	278
II.3.1.4. El operador de incremento en expresiones .	279
II.3.2. El bucle <code>for</code> en C++	281
II.3.2.1. Bucles <code>for</code> con cuerpo vacío	281
II.3.2.2. Bucles <code>for</code> con sentencias de incremento in- correctas	281
II.3.2.3. Modificación del contador	282

II.3.2.4. El bucle <code>for</code> como ciclo controlado por condición	285
II.3.3. Otras (perniciosas) estructuras de control	293

Tema I

Introducción a la Programación

Objetivos:

- ▷ Introducir los conceptos básicos de programación, para poder construir los primeros programas.
- ▷ Introducir los principales tipos de datos disponibles en C++ para representar información del mundo real.
- ▷ Enfatizar, desde un principio, la necesidad de seguir buenos hábitos de programación.

I.1. El ordenador, algoritmos y programas

I.1.1. El Ordenador: Conceptos Básicos

*"Los ordenadores son inútiles. Sólo pueden darte respuestas".
Pablo Picasso*



- ▷ **Hardware**
- ▷ **Software**
- ▷ **Usuario (User)**
- ▷ **Programador (Programmer)**

I.1.2. Datos y Algoritmos

Algoritmo (Algorithm) : es una secuencia ordenada de instrucciones que resuelve un problema concreto, atendiendo a las siguientes características:

► **Características básicas:**

- ▷ Corrección (sin errores).
- ▷ Precisión (no puede haber ambigüedad).
- ▷ Repetitividad (en las mismas condiciones, al ejecutarlo, siempre se obtiene el mismo resultado).

► **Características esenciales:**

- ▷ Finitud (termina en algún momento). Número finito de órdenes no implica finitud.
- ▷ Validez (resuelve el problema pedido)
- ▷ Eficiencia (lo hace en un tiempo aceptable)

Un **dato (data)** es una unidad de información que representamos en el ordenador (longitud del lado de un triángulo rectángulo, longitud de la hipotenusa, nombre de una persona, número de habitantes, el número π , etc)

Los algoritmos operan sobre los datos. Usualmente, reciben unos *datos de entrada* con los que operan, y a veces, calculan unos nuevos *datos de salida*.

Ejemplo. Algoritmo de la media aritmética de N valores.

- ▷ **Datos de entrada:** valor1, valor2, ..., valorN
- ▷ **Datos de salida:** media
- ▷ **Instrucciones en lenguaje natural:**
Sumar los N valores y dividir el resultado por N

Ejemplo. Algoritmo para la resolución de una ecuación de primer grado
 $ax + b = 0$

- ▷ **Datos de entrada:** a, b
- ▷ **Datos de salida:** x
- ▷ **Instrucciones en lenguaje natural:**
Calcular x como el resultado de la división $-b/a$

Podría mejorarse el algoritmo contemplando el caso de ecuaciones degeneradas, es decir, con a o b igual a cero

Ejemplo. Algoritmo para el cálculo de la hipotenusa de un triángulo rectángulo.

- ▷ **Datos de entrada:** lado1, lado2
- ▷ **Datos de salida:** hipotenusa
- ▷ **Instrucciones en lenguaje natural:**

$$\text{hipotenusa} = \sqrt{\text{lado1}^2 + \text{lado2}^2}$$

Ejemplo. Algoritmo para ordenar un vector (lista) de valores numéricos.

$$(9, 8, 1, 6, 10, 4) \longrightarrow (1, 4, 6, 8, 9, 10)$$

- ▷ **Datos de entrada:** el vector
- ▷ **Datos de salida:** el mismo vector
- ▷ **Instrucciones en lenguaje natural:**
 - Calcular el mínimo valor de todo el vector
 - Intercambiarlo con la primera posición
 - Volver a hacer lo mismo con el vector formado por todas las componentes menos la primera.

$(9, 8, 1, 6, 10, 4) \rightarrow$

$(1, 8, 9, 6, 10, 4) \rightarrow$

$(X, 8, 9, 6, 10, 4) \rightarrow$

$(X, 4, 9, 6, 10, 8) \rightarrow$

$(X, X, 9, 6, 10, 8) \rightarrow$

...

Instrucciones no válidas en un algoritmo:

- Calcular un valor *bastante* pequeño en todo el vector
- Intercambiarlo con el que está en una posición *adecuada*
- Volver a hacer lo mismo con el vector formado por *la mayor parte* de las componentes.

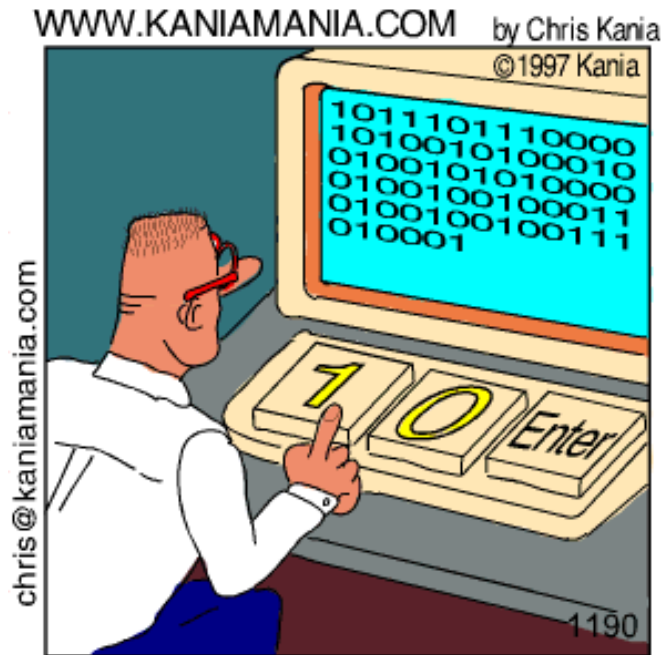
Una vez diseñado el algoritmo, debemos escribir las órdenes que lo constituyen en un lenguaje que entienda el ordenador.

"First, solve the problem. Then, write the code".



I.1.3. Lenguajes de programación

Código binario (Binary code) :



Real programmers code in binary.

"There are 10 types of people in the world, those who can read binary, and those who can't".



Lenguaje de programación (Programming language) : Lenguaje formal utilizado para comunicarnos con un ordenador e imponerle la ejecución de un conjunto de órdenes.

- ▷ **Lenguaje ensamblador (Assembly language)** . Depende del microprocesador (Intel 8086, Motorola 88000, etc) Se usa para programar drivers, microcontroladores (que son circuitos integrados que agrupan microprocesador, memoria y periféricos), compiladores, etc. Se ve en otras asignaturas.

```
.model small
.stack
.data
    Cadena1 DB 'Hola Mundo.$'
.code
    mov ax, @data
    mov ds, ax
    mov dx, offset Cadena1
    mov ah, 9
    int 21h
end
```

- ▷ **Lenguajes de alto nivel (High level language)** (C, C++, Java, Lisp, Prolog, Perl, Visual Basic, C#, Go ...) En esta asignatura usaremos **C++11/14 (ISO C++)**.

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

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

Reseña histórica del lenguaje C++:

- 1967 Martin Richards: BCPL para escribir S.O.
- 1970 Ken Thompson: B para escribir UNIX (inicial)
- 1972 Dennis Ritchie: C
- 1983 Comité Técnico X3J11: ANSI C
- 1983 Bjarne Stroustrup: C++
- 1989 Comité técnico X3J16: ANSI C++
- 1990 Internacional Standarization Organization <http://www.iso.org>
 - Comité técnico JTC1: Information Technology
 - Subcomité SC-22: Programming languages, their environments and system software interfaces.
 - Working Group 21: C++
 - <http://www.open-std.org/jtc1/sc22/wg21/>
- 2011 Revisión del estándar con importantes cambios.
- 2014 Última revisión del estándar con cambios menores.
- 2017? Actualmente en desarrollo la siguiente versión C++ 17.

¿Qué programas se han hecho en C++?

Buscador de Google, Amazon, sistema de reservas aéreas (Amadeus), omnipresente en la industria automovilística y aérea, sistemas de telecomunicaciones, el explorador Mars Rovers, el proyecto de secuenciación del genoma humano, videojuegos como Doom, Warcraft, Age of Empires, Halo, la mayor parte del software de Microsoft y una gran parte del de Apple, la máquina virtual Java, Photoshop, Thunderbird y Firefox, MySQL, OpenOffice, etc.

Implementación de un algoritmo (Algorithm implementation) : Transcripción de un algoritmo a un lenguaje de programación.

Cada lenguaje de programación tiene sus propias instrucciones. Éstas se escriben en un fichero de texto normal. Al código escrito en un lenguaje concreto se le denomina **código fuente (source code)** . En C++ llevan la extensión `.cpp`.

Ejemplo. Implementación del algoritmo para el cálculo de la media de 4 valores en C++:

```
suma = valor1 + valor2 + valor3 + valor4;  
media = suma / 4;
```

Ejemplo. Implementación del algoritmo para el cálculo de la hipotenusa:

```
hipotenusa = sqrt(lado1*lado1 + lado2*lado2);
```

Ejemplo. Implementación del algoritmo para la ordenación de un vector.

```
for (int izda = 0 ; izda < total_utilizados ; izda++){  
    minimo = vector[izda];  
    posicion_minimo = izda;  
  
    for (int i = izda + 1; i < total_utilizados ; i++){  
        if (vector[i] < minimo){  
            minimo = vector[i];  
            posicion_minimo = i;  
        }  
    }  
  
    intercambia = vector[izda];  
    vector[izda] = vector[posicion_minimo];  
    vector[posicion_minimo] = intercambia;  
}
```

Para que las instrucciones anteriores puedan ejecutarse correctamente, debemos especificar dentro del código fuente los datos con los que vamos a trabajar, incluir ciertos recursos externos, etc. Todo ello constituye un programa:

Un *programa (program)* es un conjunto de instrucciones especificadas en un lenguaje de programación concreto, que pueden ejecutarse en un ordenador.

Ejemplo. Programa para calcular la hipotenusa de un triángulo rectángulo.

Pitagoras.cpp

```
/*
    Programa simple para el cálculo de la hipotenusa
    de un triángulo rectángulo, aplicando el teorema de Pitágoras
*/

#include <iostream>    // Inclusión de recursos de E/S
#include <cmath>        // Inclusión de recursos matemáticos
using namespace std;

int main(){           // Programa Principal
    double lado1;      // Declara variables para guardar
    double lado2;      // los dos lados y la hipotenusa
    double hipotenusa;

    cout << "Introduzca la longitud del primer cateto: ";
    cin >> lado1;
    cout << "Introduzca la longitud del segundo cateto: ";
    cin >> lado2;

    hipotenusa = sqrt(lado1*lado1 + lado2*lado2);

    cout << "\nLa hipotenusa vale " << hipotenusa;
}
```

http://decsai.ugr.es/jccubero/FP/I_Pitagoras.cpp

La **programación (programming)** es el proceso de diseñar, codificar (implementar), depurar y mantener un programa.

Un programa incluirá la implementación de uno o más algoritmos.

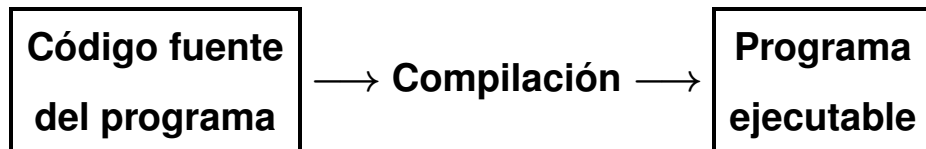
Ejemplo. Programa para dibujar planos de pisos.

Utilizará algoritmos para dibujar cuadrados, de medias aritméticas, salidas gráficas en plotter, etc.

Muchos de los programas que se verán en FP implementarán un único algoritmo.

I.1.4. Compilación

Para obtener el programa ejecutable (el fichero en binario que puede ejecutarse en un ordenador) se utiliza un *compilador (compiler)* :



La extensión en Windows de los programas ejecutables es .exe

Código Fuente: Pitagoras.cpp

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

int main() {
    double lado1;
    .....
    cout << "Introduzca la longitud ...
    cin >> lado1;
    .....
}
```



Programa Ejecutable: Pitagoras.exe

```
10011000010000
10010000111101
00110100000001
11110001011110
11100001111100
11100101011000
00001101000111
00011000111100
```


I.2. Especificación de programas

En este apartado se introducen los conceptos básicos involucrados en la construcción de un programa. Se introducen términos que posteriormente se verán con más detalle.

I.2.1. Organización de un programa

- ▷ Los programas en C++ pueden dividirse en varios ficheros aunque por ahora vamos a suponer que cada programa está escrito en un único fichero (`Pitagoras.cpp`).
- ▷ Se pueden incluir comentarios en lenguaje natural.

```
/* Comentario partido en
   varias líneas */
// Comentario en una sola línea
```

El texto de un comentario no es procesado por el compilador.

- ▷ Al principio del fichero se indica que vamos a usar una serie de recursos definidos en un fichero externo o *biblioteca (library)*

```
#include <iostream>
#include <cmath>
```

También aparece

```
using namespace std;
```

La finalidad de esta declaración se verá posteriormente.

- ▷ A continuación aparece `int main(){` que indica que comienza el programa principal. Éste se extiende desde la llave abierta `{`, hasta encontrar la correspondiente llave cerrada `}`.
- ▷ Dentro del programa principal van las sentencias. Una *sentencia*

(*sentence/statement*) es una parte del código fuente que el compilador traduce en una instrucción en código binario. Ésta van obligatoriamente separadas por punto y coma ; y se van ejecutando secuencialmente de arriba abajo. En el tema II se verá como realizar *saltos*, es decir, alterar la estructura secuencial.

- ▷ Cuando llega a la llave cerrada } correspondiente a `main()`, y si no han aparecido problemas, el programa termina de ejecutarse y el Sistema Operativo libera los recursos asignados a dicho programa.

Veamos algunos tipos de sentencias usuales:

- ▷ ***Sentencias de declaración de datos***

En la página 3 veíamos el concepto de dato. Debemos asociar cada dato a un único ***tipo de dato (data type)*** . Éste determinará los valores que podremos asignarle y las operaciones que podremos realizar.

El compilador ofrece distintos tipos de datos como por ejemplo enteros (`int`), reales (`double`), caracteres (`char`), etc. En una sentencia de ***declaración (definición) (declaration (definition))*** , el programador indica el nombre o ***identificador (identifier)*** que usará para referirse a un dato concreto y establece su tipo de dato, el cual no se podrá cambiar posteriormente.

Cada dato que se desee usar en un programa debe declararse previamente. Por ahora, lo haremos al principio (después de `main`). En el siguiente ejemplo, declaramos tres datos de tipo real:

```
double lado1;  
double lado2;  
double hipotenusa;
```

También pueden declararse en una única línea, separándolas con una coma:

```
double lado1, lado2, hipotenusa;
```

▷ **Sentencias de asignación**

A los datos se les asigna un **valor (value)** a través del denominado **operador de asignación (assignment operator)** = (no confundir con la igualdad en Matemáticas)

```
lado1 = 7;
lado2 = 5;
hipotenusa = sqrt(lado1*lado1 + lado2*lado2);
```

Asigna 7 a lado1, 5 a lado2 y asigna al dato hipotenusa el resultado de evaluar lo que aparece a la derecha de la asignación. Se ha usado el **operador (operator)** de multiplicación (*), el operador de suma (+) y la **función (function)** raíz cuadrada (sqrt). Posteriormente se verá con más detalle el uso de operadores y funciones.

Podremos cambiar el valor de los datos tantas veces como queramos.

```
lado1 = 7;    // lado1 contiene 7
lado1 = 8;    // lado1 contiene 8. Se pierde el antiguo valor (7)
```

▷ **Sentencias de entrada de datos**

¿Y si queremos asignarle a lado1 un valor introducido por el usuario del programa?

Las sentencias de **entrada de datos (data input)** permiten leer valores desde el dispositivo de entrada establecido por defecto. Por ahora, será el teclado (también podrá ser un fichero, por ejemplo). Se construyen usando cin, que es un recurso externo incluido en la biblioteca iostream. Por ejemplo, al ejecutarse la sentencia

```
cin >> lado1;
```

el programa espera a que el usuario introduzca un valor real desde el teclado (dispositivo de entrada) y, cuando se pulsa la tecla Intro, lo almacena en el dato lado1 (si la entrada es desde un fichero, no hay que introducir Intro). Conforme se va escribiendo el valor, éste se muestra en pantalla, incluyendo el salto de línea.

La lectura de datos con `cin` puede considerarse como una asignación en tiempo de ejecución

▷ **Sentencias de salida de datos**

Por otra parte, las sentencias de **salida de datos (data output)** permiten escribir mensajes y los valores de los datos en el dispositivo de salida establecido por defecto. Por ahora, será la pantalla (podrá ser también un fichero). Se construyen usando `cout`, que es un recurso externo incluido en la biblioteca `iostream`.

```
cout << "Este texto se muestra tal cual " << dato;
```

- Lo que haya dentro de un par de comillas dobles se muestra tal cual, excepto los caracteres precedidos de `\`. Por ejemplo, `\n` hace que el cursor salte al principio de la línea siguiente.

```
cout << "Bienvenido. Salto a la siguiente linea.\n";  
cout << "\nEmpiezo en una nueva linea.";
```

Mostraría en pantalla lo siguiente:

```
Bienvenido. Salto a la siguiente linea.  
Empiezo en una nueva linea.
```

- Los números se escriben tal cual (decimales con punto)

```
cout << 3.1415927;
```

- Si ponemos un dato, se imprime su contenido.

```
cout << hipotenusa;
```

Podemos usar incluir en la misma sentencia la salida de mensajes con la de datos, separándolos con `<<`:

```
cout << "\nLa hipotenusa vale " << hipotenusa;
```

Si no hubiésemos incluido `using namespace std;` al inicio del programa, habría que anteponer el **espacio de nombres (namespace)** `std` en la llamada a `cout` y `cin` de la siguiente forma:

```
std::cout << variable;
```

Los namespaces sirven para organizar los recursos (funciones, clases, etc) ofrecidos por el compilador o contruidos por nosotros. La idea es similar a la estructura en carpetas de los ficheros de un sistema operativo. En FP no crearemos espacios de nombres; simplemente usaremos el estándar (`std`)

Como resumen, podemos decir que la estructura básica de un programa quedaría de la forma siguiente (los corchetes delimitan secciones opcionales):

```
[ /* Breve descripción en lenguaje natural
    de lo que hace el programa */ ]
[ Inclusión de recursos externos ]
[ using namespace std; ]
int main(){
    [ Declaración de datos ]
    [ Sentencias del programa separadas por ; ]
}
```

I.2.2. Elementos básicos de un lenguaje de programación

I.2.2.1. Tokens y reglas sintácticas

A la hora de escribir un programa, cada lenguaje de programación tiene una sintaxis propia que debe respetarse. Ésta queda definida por:

- a) Los **componentes léxicos (tokens)** . Formados por caracteres alfanuméricos y/o simbólicos. Representan la unidad léxica mínima que el lenguaje entiende.

58 main ; (== = hipotenusa * /*

Pero por ejemplo, ni ; ni ((* ni / * son tokens válidos.

- b) **Reglas sintácticas (Syntactic rules)** : determinan cómo han de combinarse los tokens para formar sentencias. Algunas se especifican con tokens especiales (formados usualmente por símbolos):

- Separador de sentencias ;
- Para agrupar varias sentencias se usa { }
- Se verá su uso en el tema II. Por ahora, sirve para agrupar las sentencias que hay en el programa principal
- Para agrupar expresiones (fórmulas) se usa ()
- sqrt((lado1*lado1) + (lado2*lado2));

I.2.2.2. Palabras reservadas

Suelen ser tokens formados por caracteres alfabéticos.

Tienen un significado específico para el compilador, y por tanto, el programador no puede definir datos con el mismo identificador.

Algunos usos:

- ▷ `main` (formalmente, `main` no es una palabra reservada, pero a efectos prácticos, así lo consideraremos)
- ▷ Para definir tipos de datos como por ejemplo `double`
- ▷ Para establecer el *flujo de control (control flow)*, es decir, para especificar el orden en el que se han de ejecutar las sentencias, como `if`, `while`, `for` etc.

Estos se verán en el tema II.

Palabras reservadas comunes a C (C89) y C++

<code>auto</code>	<code>break</code>	<code>case</code>	<code>char</code>	<code>const</code>	<code>continue</code>	<code>default</code>
<code>do</code>	<code>double</code>	<code>else</code>	<code>enum</code>	<code>extern</code>	<code>float</code>	<code>for</code>
<code>goto</code>	<code>if</code>	<code>int</code>	<code>long</code>	<code>register</code>	<code>return</code>	<code>short</code>
<code>signed</code>	<code>sizeof</code>	<code>static</code>	<code>struct</code>	<code>switch</code>	<code>typedef</code>	<code>union</code>
<code>unsigned</code>	<code>void</code>	<code>volatile</code>	<code>while</code>			

Palabras reservadas adicionales de C++

<code>and</code>	<code>and_eq</code>	<code>asm</code>	<code>bitand</code>	<code>bitor</code>	<code>bool</code>	<code>catch</code>
<code>class</code>	<code>compl</code>	<code>const_cast</code>	<code>delete</code>	<code>dynamic_cast</code>	<code>explicit</code>	<code>export</code>
<code>false</code>	<code>friend</code>	<code>inline</code>	<code>mutable</code>	<code>namespace</code>	<code>new</code>	<code>not</code>
<code>not_eq</code>	<code>operator</code>	<code>or</code>	<code>or_eq</code>	<code>private</code>	<code>protected</code>	<code>public</code>
<code>reinterpret_cast</code>	<code>static_cast</code>	<code>template</code>	<code>this</code>	<code>throw</code>	<code>true</code>	<code>try</code>
<code>typeid</code>	<code>typename</code>	<code>using</code>	<code>virtual</code>	<code>wchar_t</code>	<code>xor</code>	<code>xor_eq</code>

I.2.3. Tipos de errores en la programación

▷ *Errores en tiempo de compilación (Compilation error)*

Ocasionados por un fallo de sintaxis en el código fuente.

No se genera el programa ejecutable.

```
/* CONTIENE ERRORES */
#include <iostream am>

using namespace std;

int main(){
    double main;
    double lado1:
    double lado 2,
    double hipotenusa:

    2 = lado1;
    lado1 = 2
    hipotenusa = sqrt(lado1*lado1 + lado2*lado2);
    cout << "La hipotenusa vale << hipotenusa;
)
```


"Software and cathedrals are much the same. First we build them, then we pray".



▷ **Errores en tiempo de ejecución (Execution error)**

Se ha generado el programa ejecutable, pero se produce un error durante la ejecución. El programa aborta su ejecución.

```
int dato_entero;  
int otra_variable;  
  
dato_entero = 0;  
otra_variable = 7 / dato_entero;
```



Cuando dividimos un dato entero por cero, se produce un error en tiempo de ejecución.

▷ **Errores lógicos (Logic errors)**

Se ha generado el programa ejecutable, pero el programa ofrece una solución equivocada.

```
.....  
lado1 = 4;  
lado2 = 9;  
hipotenusa = sqrt(lado1+lado1 + lado2*lado2);  
.....
```

I.2.4. Cuidando la presentación

Además de generar un programa sin errores, debemos asegurar que:

- ▷ El código fuente sea fácil de leer por otro programador.
- ▷ El programa sea fácil de manejar por el usuario.

I.2.4.1. Escritura de código fuente

A lo largo de la asignatura veremos normas que tendremos que seguir para que el código fuente que escribamos sea fácil de leer por otro programador. Debemos usar espacios y líneas en blanco para separar tokens y grupos de sentencias. El compilador ignora estos separadores pero ayudan en la lectura del código fuente.

Para hacer más legible el código fuente, usaremos separadores como el espacio en blanco, el tabulador y el retorno de carro



Este código fuente genera el mismo programa que el de la página 11 pero es mucho más difícil de leer.

```
#include<iostream>#include<cmath>using namespace std;
int main(){
    double lado1;double lado2;double hipotenusa;
    cout<<"Introduzca la longitud del primer cateto: ";
    cin>>lado1;
    cout<<"Introduzca la longitud del segundo cateto: ";cin>>lado2;
    hipotenusa=sqrt(lado1*lado1+lado2*lado2);cout<<"\nLa hipotenusa vale "
    <<hipotenusa;
}
```



I.2.4.2. Etiquetado de las Entradas/Salidas

Es importante dar un formato adecuado a la salida de datos en pantalla. El usuario del programa debe entender claramente el significado de todas sus salidas.

```
totalVentas = 45;  
numeroVentas = 78;  
cout << totalVentas << numeroVentas; // Imprime 4578
```



```
cout << "\nSuma total de ventas = " << totalVentas;  
cout << "\nNúmero total de ventas = " << numeroVentas;
```



Las entradas de datos también deben etiquetarse adecuadamente:

```
cin >> lado1;  
cin >> lado2;
```



```
cout << "Introduzca la longitud del primer cateto: ";  
cin >> lado1;  
cout << "Introduzca la longitud del segundo cateto: ";  
cin >> lado2;
```



I.3. Datos y tipos de datos

I.3.1. Representación en memoria de datos e instrucciones

Tanto las instrucciones como los datos son combinaciones adecuadas de 0 y 1.

<i>Datos</i>									
"Juan Pérez"	→ <table><tr><td>1</td><td>0</td><td>1</td><td>1</td><td>..</td><td>0</td><td>1</td><td>1</td></tr></table>	1	0	1	1	..	0	1	1
1	0	1	1	..	0	1	1		
75225813	→ <table><tr><td>1</td><td>1</td><td>0</td><td>1</td><td>..</td><td>0</td><td>0</td><td>0</td></tr></table>	1	1	0	1	..	0	0	0
1	1	0	1	..	0	0	0		
3.14159	→ <table><tr><td>0</td><td>0</td><td>0</td><td>1</td><td>..</td><td>1</td><td>1</td><td>1</td></tr></table>	0	0	0	1	..	1	1	1
0	0	0	1	..	1	1	1		
<i>Instrucciones</i>									
Abrir Fichero	→ <table><tr><td>0</td><td>0</td><td>0</td><td>1</td><td>..</td><td>1</td><td>1</td><td>1</td></tr></table>	0	0	0	1	..	1	1	1
0	0	0	1	..	1	1	1		
Imprimir	→ <table><tr><td>1</td><td>1</td><td>0</td><td>1</td><td>..</td><td>0</td><td>0</td><td>0</td></tr></table>	1	1	0	1	..	0	0	0
1	1	0	1	..	0	0	0		

Nos centramos en los datos.

I.3.2. Datos y tipos de datos

I.3.2.1. Declaración de datos

Al trabajar con un lenguaje de alto nivel, no haremos referencia a la secuencia de 0 y 1 que codifican un valor concreto, sino a lo que representa para nosotros.

Nombre de empleado: Juan Pérez

Número de habitantes : 75225813

π : 3.14159

Cada lenguaje de programación ofrece sus propios tipos de datos, denominados *tipos de datos primitivos (primitive data types)* . Por ejemplo, en C++: `string`, `int`, `double`, etc. Además, el programador podrá crear sus propios tipos usando otros recursos, como por ejemplo, las *clases* (ver tema IV).

Cuando se declara un dato, el compilador reserva una zona de memoria para trabajar con ella. Ningún otro dato podrá usar dicha zona.

```
string nombre_empleado;  
int    numero_habitantes;  
double Pi;  
  
nombre_empleado = "Pedro Ramírez";  
nombre_empleado = "Juan Pérez";    // Se pierde el antiguo  
nombre_empleado = 37;              // Error de compilación  
numero_habitantes = "75225";       // Error de compilación  
numero_habitantes = 75225;  
Pi = 3.14156;  
Pi = 3.1415927;                    // Se pierde el antiguo
```

nombre_empleado	numero_habitantes	Pi
Juan Pérez	75225	3.1415927

Opcionalmente, se puede dar un valor inicial durante la declaración.

```
<tipo> <identificador> = <valor_inicial>;
```

```
double dato = 4.5;
```

produce el mismo resultado que:

```
double dato;  
dato = 4.5;
```

Recuerde que cada dato necesita un identificador único. Un identificador de un dato es un token formado por caracteres alfanuméricos con las siguientes restricciones:

- ▷ Debe empezar por una letra o subrayado (`_`)
- ▷ No pueden contener espacios en blanco ni ciertos caracteres especiales como letras acentuadas, la letra ñe, las barras `\` o `/`, etc.
Ejemplo: `lado1` `lado2` `precio_con_IVA`
- ▷ El compilador determina la máxima longitud que pueden tener (por ejemplo, 31 caracteres)
- ▷ Sensible a mayúsculas y minúsculas.
`lado` y `Lado` son dos identificadores distintos.
- ▷ No se podrá dar a un dato el nombre de una palabra reservada.

```
#include <iostream>
using namespace std;
int main(){
    double long;    // Error de sintaxis
    double cout;    // Error de sintaxis
```

- ▷ No es recomendable usar el nombre de algún identificador utilizado en las *bibliotecas estándar* (por ejemplo, `cout`)

```
// #include <iostream>
int main(){
    double cout;    // Evítelo
```

Ejercicio. Determine cuáles de los siguientes son identificadores válidos. Si son inválidos explicar por qué.

- a) `registro1` b) `lregistro` c) `archivo_3` d) `main`
e) `nombre y direccion` f) `dirección` g) `diseño`

Hemos visto las restricciones impuestas por el compilador para elegir el nombre de un dato. Además, debemos seguir otras normas que faciliten la legibilidad de nuestros programas:

- ▷ El identificador de un dato debe reflejar su semántica (contenido). Por eso, salvo excepciones (como las variables contadoras de los bucles -tema II-) no utilizaremos nombres con pocos caracteres:



v, l1, l2, hp



voltaje, lado1, lado2, hipotenusa

- ▷ No utilizaremos nombres genéricos



aux, vector1



copia, calificaciones

- ▷ Usaremos minúsculas para los datos variables. Las constantes se escribirán en mayúsculas. Los nombres compuestos se separarán con subrayado _:



precioventapublico, tasaanual



precio_venta_publico, tasa_anual

Este es el denominado *estilo snake case* . Hay otros como *estilo camelCase* o *estilo UpperCamelCase* . Este último es el que usaremos para las funciones y métodos.

- ▷ Podremos usar siglas como identificadores, siempre que sean ampliamente conocidas.



pvp, tae

- ▷ Evitaremos, en la medida de lo posible, nombrar datos que difieran únicamente en la capitalización, o un sólo carácter.



cuenta, cuentas, Cuenta



cuenta, coleccion_cuentas, cuenta_ppal

Hay una excepción a esta norma. Cuando veamos las clases, podremos definir una clase con el nombre `Cuenta` y una instancia de dicha clase con el nombre `cuenta`.

Como programadores profesionales, debemos seguir las normas de codificación de código establecidas en la empresa.



Cada empresa utilizará sus propias directrices de codificación. Es importante seguirlas escrupulosamente para facilitar la lectura de código escrito por empleados distintos.

Nosotros seguiremos, en parte, las directrices indicadas por Google en *Google C++ Style Guide*

<https://google.github.io/styleguide/cppguide.html>

I.3.2.2. Inicialización de los datos

Cuando se declara un dato y no se inicializa, éste no tiene ningún valor asignado *por defecto*. El valor almacenado es *indeterminado* y puede variar de una ejecución a otra del programa. Lo representaremos gráficamente por ?

Ejemplo. Calcule la cuantía de la retención a aplicar sobre el sueldo de un empleado, sabiendo el porcentaje de ésta.

```
/*
    Cálculo de la retención a aplicar en el salario bruto
*/
#include <iostream>
using namespace std;

int main(){
    double salario_bruto; // Salario bruto, en euros
    double retencion;     // Retención -> parte del salario bruto
                        // que se queda Hacienda

    cout << "Introduzca salario bruto: ";
    cin >> salario_bruto; // El usuario introduce 32538

    retencion = salario_bruto * 0.18;
    cout << "\nRetención a aplicar: " << retencion;
}
```

salario_bruto	retencion
?	?

salario_bruto	retencion
32538.0	4229.94

Un error **lógico** muy común es usar dentro de una expresión un dato no asignado:

```
int main(){  
    double salario_bruto;  
    double retencion;  
  
    retencion = salario_bruto * 0.18;    // salario_bruto indeterminado  
  
    cout << "Retención a aplicar: " << retencion;  
}
```



Imprimirá un valor indeterminado.

GLGO: Garbage input, garbage output



I.3.2.3. Literales

Un *literal (literal)* es la especificación de un valor concreto de un tipo de dato. Dependiendo del tipo, tenemos:

- ▷ *Literales numéricos (numeric literals)* : son tokens numéricos.
Para representar datos reales, se usa el punto . para especificar la parte decimal:
2 3 3.1415927
- ▷ *Literales de cadenas de caracteres (string literals)* : Son cero o más caracteres encerrados entre comillas dobles:
"Juan Pérez"
- ▷ Literales de otros tipos, como *literales de caracteres (character literals)* 'a', '!', etc, *literales lógicos (boolean literals)* true, etc.

I.3.2.4. Datos constantes

Podríamos estar interesados en usar datos a los que sólo permitimos tomar un único valor, fijado de antemano. Es posible con una *constante (constant)* . Se declaran como sigue:

```
const <tipo> <identif> = <expresión>;
```

- ▷ A los datos no constantes se les denomina *variables (variables)* .
- ▷ A las constantes se les aplica las mismas consideraciones que hemos visto sobre tipo de dato y reserva de memoria.
- ▷ Los identificadores de las constantes suelen ser sólo en mayúsculas para diferenciarlos de las variables.

Ejemplo. Calcule la longitud de una circunferencia y el área de un círculo, sabiendo su radio.

```
/*
    Programa que pide el radio de una circunferencia
    e imprime su longitud y el área del círculo
*/
#include <iostream>
using namespace std;

int main() {
    const double PI = 3.1416;
    double area, radio, longitud;

    // PI = 3.15;    <- Error de compilación 😊

    cout << "Introduzca el valor del radio ";
    cin >> radio;

    area = PI * radio * radio;
    longitud = 2 * PI * radio;

    cout << "\nEl área del círculo es: " << area;
    cout << "\nLa longitud de la circunferencia es: " << longitud;
}
```

http://decsai.ugr.es/jccubero/FP/I_Circunferencia.cpp

Compare el anterior código con el siguiente:

```
.....
area = 3.1416 * radio * radio;
longitud = 2 * 3.1416 * radio;
```



Ventajas al usar constantes en vez de literales:

- ▷ El nombre dado a la constante (π) proporciona más información al programador y hace que el código sea más legible.

Esta ventaja podría haberse conseguido usando un dato variable, pero entonces podría cambiarse su valor por error dentro del código. Al ser constante, su modificación no es posible.

- ▷ Código menos propenso a errores. Para cambiar el valor de π (a 3.1415927, por ejemplo), sólo hay que modificar la línea de la declaración de la constante.

Si hubiésemos usado literales, tendríamos que haber recurrido a un cut-paste, muy propenso a errores.

Evite siempre el uso de *números mágicos* (*magic numbers*), es decir, literales numéricos cuyo significado en el código no queda claro.

Fomentaremos el uso de datos constantes en vez de literales para representar toda aquella información que sea constante durante la ejecución del programa.

IMPORTANT

No debemos llevar al extremo la anterior recomendación:

```
.....  
const int DOS = 2;  
.....  
longitud = DOS * PI * radio;
```



Ejemplo. Modifique el ejemplo de la página 31 evitando el uso de números mágicos. Para ello, introduzca una constante que contenga el valor de la fracción de retención (0.18).

```
/*
    Cálculo de la retención a aplicar en el salario bruto
*/
#include <iostream>
using namespace std;

int main(){
    const double FRACCION_RETENCION = 0.18; // Fracción de retención (18%)
    double salario_bruto; // Salario bruto, en euros
    double retencion;      // Retención -> parte del salario bruto
                           // que se queda Hacienda

    salario_bruto  FRACCION_RETENCION  retencion
    [?]           [0.18]                [?]

    cout << "Introduzca salario bruto: ";
    cin >> salario_bruto;

    retencion = salario_bruto * FRACCION_RETENCION;

    cout << "\nRetención a aplicar: " << retencion;
}
```

salario_bruto	FRACCION_RETENCION	retencion
32538.0	0.18	4229.94

http://decsai.ugr.es/jccubero/FP/I_retencion.cpp

Ejemplo. Retome el ejemplo anterior y calcule el *salario neto (net income)* . Éste se define como el *salario bruto (gross income)* menos la retención.

```
int main(){
    .....
    retencion = salario_bruto * FRACCION_RETENCION;
    salario_netto = salario_bruto - retencion;

    cout << "\nRetención a aplicar: " << retencion;
    cout << "\nSalario neto      : " << salario_netto;
}
```

Otra forma de trabajar cuando hay que aplicar subidas o bajadas porcentuales es utilizar un *índice de variación* . En el ejemplo anterior, para calcular el salario neto, basta multiplicar por el índice de variación 0.82. Una vez calculado el salario neto, procederíamos a calcular la retención:

```
int main(){
    const double IV_SALARIO = 1 - 0.18;
                                // Índice de variación (Resta un 18%)
    .....
    salario_netto = salario_bruto * IV_SALARIO;
    retencion = salario_bruto - salario_netto;

    cout << "\nRetención a aplicar: " << retencion;
    cout << "\nSalario neto      : " << salario_netto;
```

Nota:

Si tuviésemos que aplicar un incremento porcentual en vez de un decremento, el índice de variación será mayor que uno. Por ejemplo, una subida del 18% equivale a multiplicar por 1.18

Observe que, para que quede más claro, dejamos la expresión $1 - 0.18$ en la definición de la constante.

Si el valor de una constante se obtiene a partir de una expresión, no evalúe manualmente dicha expresión: incluya la expresión completa en la definición de la constante.

Una sintaxis de programa algo más completa:

```
[ /* Breve descripción en lenguaje natural  
    de lo que hace el programa */ ]  
  
[ Inclusión de recursos externos ]  
[ using namespace std; ]  
  
int main(){  
    [Declaración de constantes]  
    [Declaración de variables]  
  
    [Sentencias del programa separadas por ;]  
}
```

I.3.2.5. Ámbito de un dato

El **ámbito** (*scope*) de un dato (variable o constante) v es el conjunto de todos aquellos sitios que pueden acceder a v .

El ámbito depende del lugar en el que se declara el dato.

Por ahora, todos los datos los estamos declarando dentro del programa principal y su ámbito es el propio programa principal. En temas posteriores veremos otros sitios en los que se pueden declarar datos y por tanto habrá que analizar cuál es su ámbito.

I.4. Operadores y expresiones

I.4.1. Expresiones

Una **expresión** (*expression*) es una combinación de datos y operadores sintácticamente correcta, que devuelve un valor. El caso más sencillo de expresión es un literal o un dato:

```
3
lado1
```

La aplicación de un operador sobre uno o varios datos es una expresión:

```
3 + 5
lado1 * lado1
lado2 * lado2
```

En general, los operadores y funciones se aplican sobre expresiones y el resultado es una expresión:

```
lado1 * lado1 + lado2 * lado2
sqrt(lado1 * lado1 + lado2 * lado2)
```

Una expresión **NO** es una sentencia de un programa:

- ▷ **Expresión:** `sqrt(lado1*lado1 + lado2*lado2)`
- ▷ **Sentencia:** `hipotenusa = sqrt(lado1*lado1 + lado2*lado2);`

Las expresiones pueden aparecer a la derecha de una asignación, pero no a la izquierda.

```
3 + 5 = lado1; // Error de compilación
```

Todo aquello que puede aparecer a la izquierda de una asignación se

conoce como *l-value* (left) y a la derecha *r-value* (right)

Cuando el compilador evalúa una expresión, devuelve un valor de un tipo de dato (entero, real, carácter, etc.). Diremos que la expresión es de dicho tipo de dato. Por ejemplo:

$3 + 5$ es una expresión entera

$3.5 + 6.7$ es una expresión real

Cuando se usa una expresión dentro de `cout`, el compilador detecta el tipo de dato resultante y la imprime de forma adecuada.

```
cout << "\nResultado = " << 3 + 5;  
cout << "\nResultado = " << 3.5 + 6.7;
```

Imprime en pantalla:

```
Resultado = 8  
Resultado = 10.2
```

A lo largo del curso justificaremos que es mejor no incluir expresiones dentro de las instrucciones `cout` (más detalles en la página 112). Mejor guardamos el resultado de la expresión en una variable y mostramos la variable:

```
suma = 3.5 + 6.7;  
cout << "\nResultado = " << suma;
```

Evite la evaluación de expresiones en una instrucción de salida de datos. Éstas deben limitarse a imprimir mensajes y el contenido de las variables.

I.4.2. Terminología en Matemáticas

Notaciones usadas con los operadores matemáticos:

- ▷ **Notación prefija (Prefix notation)** . El operador va antes de los argumentos. Estos suelen encerrarse entre paréntesis.

$\text{seno}(3)$, $\text{tangente}(x)$, $\text{media}(\text{valor1}, \text{valor2})$

- ▷ **Notación infija (Infix notation)** . El operador va entre los argumentos.

$3 + 5$ x/y

Según el número de argumentos, diremos que un operador es:

- ▷ **Operador unario (Unary operator)** . Sólo tiene un argumento:

$\text{seno}(3)$, $\text{tangente}(x)$

- ▷ **Operador binario (Binary operator)** . Tienes dos argumentos:

$\text{media}(\text{valor1}, \text{valor2})$, $3 + 5$, x/y

- ▷ **Operador n-ario (n-ary operator)** . Tiene más de dos argumentos.

I.4.3. Operadores en Programación

Los lenguajes de programación proporcionan operadores que permiten manipular los datos.

- ▷ Se denotan a través de tokens alfanuméricos o simbólicos.
- ▷ Suelen devolver un valor.

Tipos de operadores:

- ▷ Los definidos en el núcleo del compilador.

No hay que incluir ninguna biblioteca

Suelen usarse tokens simbólicos para su representación:

+ (suma), - (resta), * (producto), etc.

Los operadores binarios suelen ser infijos:

`3 + 5`

`lado * lado`

- ▷ Los definidos en bibliotecas externas.

Por ejemplo, `cmath`

Suelen usarse tokens alfanuméricos para su representación:

`sqrt` (raíz cuadrada), `sin` (seno), `pow` (potencia), etc.

Suelen ser prefijos. Si hay varios argumentos se separan por una coma.

`sqrt(4.2)`

`sin(6.4)`

`pow(3 , 6)`

Tradicionalmente se usa el término *operador (operator)* a secas para denotar los primeros, y el término *función (function)* para los segundos. Los argumentos de las funciones se denominan *parámetros (parameter)*

Una misma variable puede aparecer a la derecha y a la izquierda de una asignación:

```
double dato;
```

```
dato = 4;           // dato contiene 4
```

```
dato = dato + 3;    // dato contiene 7
```

En una sentencia de asignación

```
variable = <expresión>
```

primero se evalúa la expresión que aparece a la derecha y luego se realiza la asignación.

I.5. Tipos de datos simples en C++

El comportamiento de un tipo de dato viene dado por:

- ▷ El **rango** (*range*) de valores que puede representar, que depende de la cantidad de memoria que dedique el compilador a su representación interna. Intuitivamente, cuanta más memoria se dedique para un tipo de dato, mayor será el número de valores que podremos representar.
- ▷ El conjunto de operadores que pueden aplicarse a los datos de ese tipo.

A lo largo de este tema se verán operadores y funciones aplicables a los distintos tipos de datos. No es necesario aprenderse el nombre de todos ellos pero sí saber cómo se usan.

Veamos qué rango tienen y qué operadores son aplicables a los tipos de datos más usados en C++. Empezamos con los enteros.

I.5.1. Los tipos de datos enteros

I.5.1.1. Representación de los enteros

Propiedad fundamental: Cualquier entero puede descomponerse como la suma de determinadas potencias de 2.

$$53 = 0*2^{15} + 0*2^{14} + 0*2^{13} + 0*2^{12} + 0*2^{11} + 0*2^{10} + 0*2^9 + 0*2^8 + 0*2^7 + \\ + 0*2^6 + 1*2^5 + 1*2^4 + 0*2^3 + 1*2^2 + 0*2^1 + 1*2^0$$

La representación en binario sería la secuencia de los factores (1,0) que acompañan a las potencias:

0000000000110101

Esta representación de un entero es independiente del lenguaje de programación. Se conoce como **bit** a la aparición de un valor 0 o 1. Un **byte** es una secuencia de 8 bits.

¿Cuántos datos distintos podemos representar?

- ▷ Dos elementos a combinar: 1, 0
- ▷ r posiciones. Por ejemplo, $r = 16$
- ▷ Se permiten repeticiones e importa el orden

$$0000000000110101 \neq 0000000000110110$$

- ▷ Por tanto, el número de datos distintos representables es 2^r

I.5.1.2. Rango de los enteros

El rango de un **entero** (*integer*) es un subconjunto del conjunto matemático \mathbb{Z} . La cardinalidad dependerá del número de bits (r) que cada compilador utilice para su almacenamiento.

Los compiladores suelen ofrecer distintos tipos enteros. En C++: `short`, `int`, `long`, etc. El más usado es `int`.

El estándar de C++ no obliga a los compiladores a usar un tamaño determinado. Lo usual es que un `int` ocupe 32 bits. El rango sería:

$$\left[-\frac{2^{32}}{2}, \frac{2^{32}}{2} - 1 \right] = [-2\,147\,483\,648, 2\,147\,483\,647]$$

```
int entero;  
entero = 53;
```

Cuando necesitemos un entero mayor, podemos usar el tipo `long long int`. También puede usarse la forma abreviada del nombre: `long long`. Es un entero de 64 bits y es estándar en C++ 11. El rango sería:

$$[-9\,223\,372\,036\,854\,775\,808, 9\,223\,372\,036\,854\,775\,807]$$

Así pues, con este tipo de dato tenemos la garantía de representar cualquier número entero de 18 cifras, positivo o negativo.

Algunos compiladores ofrecen tipos propios. Por ejemplo, Visual C++ ofrece `__int64`.

I.5.1.3. Literales enteros

Como ya vimos en la página 33, un literal es la especificación (dentro del código fuente) de un valor concreto de un tipo de dato. Los *literales enteros (integer literals)* se construyen con tokens formados por símbolos numéricos. Pueden empezar con un signo -

53 -406778 0

Nota. En el código se usa el sistema decimal (53) pero internamente, el ordenador usa el código binario (000000000110101)

Para representar un literal entero, el compilador usará el tipo `int`. Si es un literal que no cabe en un `int`, se usará otro tipo entero mayor. Por lo tanto:

- ▷ -1 es un literal de tipo `int`
- ▷ 53 es un literal de tipo `int`
- ▷ 123456789123456 es un literal de tipo `long long`

I.5.1.4. Operadores

Operadores binarios

Los más usuales son:

+ - * / %

suma, resta, producto, división entera y módulo: devuelven un entero.
Son binarios de notación infija.

El operador módulo (%) representa el resto de la división entero $a*b$

```
int n;
n = 5 * 7;      // Asigna a la variable n el valor 35
n = n + 1;     // Asigna a la variable n el valor 36
n = 25 / 9;     // Asigna a la variable n el valor 2
n = 25 % 9;     // Asigna a la variable n el valor 7
n = 5 / 7;      // Asigna a la variable n el valor 0
n = 7 / 5;      // Asigna a la variable n el valor 1
n = 173 / 10;   // Asigna a la variable n el valor 17
n = 5 % 7;      // Asigna a la variable n el valor 5
n = 7 % 5;      // Asigna a la variable n el valor 2
n = 173 % 10;   // Asigna a la variable n el valor 3
5 / 7 = n;      // Sentencia Incorrecta.
```

Operaciones usuales:

- ▷ **Extraer el dígito menos significativo:** $5734 \% 10 \rightarrow 4$
- ▷ **Truncar desde el dígito menos significativo:** $5734 / 10 \rightarrow 573$

Ejercicio. Lea un entero desde teclado que represente número de segundos y calcule el número de minutos que hay en dicha cantidad y el número de segundos restantes. Por ejemplo, en 123 segundos hay 2 minutos y 3 segundos.

Nota:

Todos los operadores que hemos visto pueden utilizarse de una forma especial junto con la asignación. Cuando una misma variable aparece a la derecha e izquierda de una asignación, la sentencia:

$$\langle \text{variable} \rangle = \langle \text{variable} \rangle \langle \text{operador} \rangle \langle \text{dato} \rangle$$

es equivalente a la siguiente:

$$\langle \text{variable} \rangle \langle \text{operador} \rangle = \langle \text{dato} \rangle$$

Algunos ejemplos de sentencias equivalentes:

$$n = n + 1; \quad n += 1;$$
$$n = n * 6; \quad n *= 6;$$
$$n = n / p; \quad n /= p;$$

Por ahora, para no aumentar la complejidad de las expresiones, evitaremos su uso. Sí lo utilizaremos cuando veamos los vectores en el tema III.

Operadores unarios de incremento y decremento

++ y -- Unarios de notación postfija.

Incrementan y decrementan, respectivamente, el valor de la variable entera sobre la que se aplican (no pueden aplicarse sobre una expresión).

```
<variable>++;    /* Incrementa la variable en 1
                  Es equivalente a:
                  <variable> = <variable> + 1; */
<variable>--;    /* Decrementa la variable en 1
                  Es equivalente a:
                  <variable> = <variable> - 1; */
```

Por ejemplo:

```
int dato = 4;
dato = dato+1;    // Asigna 5 a dato
dato++;          // Asigna 6 a dato
```

También existe una versión prefija de estos operadores. Lo veremos en el siguiente tema y analizaremos en qué se diferencian.

Operador unario de cambio de signo

- Unario de notación prefija.

Cambia el signo de la variable sobre la que se aplica.

```
int dato = 4, dato_cambiado;
dato_cambiado = -dato;    // Asigna -4 a dato_cambiado
dato_cambiado = -dato_cambiado;    // Asigna 4 a dato_cambiado
```

I.5.1.5. Expresiones enteras

Son aquellas expresiones, que al evaluarlas, devuelven un valor entero.

```
entera      56      (entera/4 + 56)%3
```

El orden de evaluación depende de la *precedencia* de los operadores.

Reglas de precedencia:

```
( )  
- (operador unario de cambio de signo)  
* / %  
+ -
```

Cualquier operador de una fila superior tiene más prioridad que cualquiera de la fila inferior.

```
variable = 3 + 5 * 7; // equivale a 3 + (5 * 7)
```

Los operadores de una misma fila tienen la misma prioridad. En este caso, para determinar el orden de evaluación se recurre a otro criterio, denominado *asociatividad (associativity)*. Puede ser de izquierda a derecha (LR) o de derecha a izquierda (RL).

```
variable = 3 / 5 * 7; // / y * tienen la misma precedencia.  
// Asociatividad LR. Equivale a (3 / 5) * 7  
variable = - -5;      // Asociatividad RL. Equivale a - (-5)
```

Ante la duda, fuerce la evaluación deseada mediante la utilización de paréntesis:

```
dato = 3 + (5 * 7);      // 38  
dato = (3 + 5) * 7;      // 56
```

Ejercicio. Teniendo en cuenta el orden de precedencia de los operadores, indique el orden en el que se evaluarían las siguientes expresiones:

a) $a + b * c - d$ **b)** $a * b / c$ **c)** $a * c \% b - d$

Ejercicio. Incremente el salario en 100 euros y calcule el número de billetes de 500 euros a usar en el pago de dicho salario.

I.5.2. Los tipos de datos reales

Un dato de tipo *real (real)* tiene como rango un subconjunto *finito* de R

- ▷ Parte entera de 4.56 → 4
- ▷ Parte real de 4.56 → 56

C++ ofrece distintos tipos para representar valores reales. Principalmente, `float` (usualmente 32 bits) y `double` (usualmente 64 bits).

```
double valor_real;  
valor_real = 541.341;
```

I.5.2.1. Literales reales

Son tokens formados por dígitos numéricos y con un único punto que separa la parte decimal de la real. Pueden llevar el signo - al principio.

800.457 4.0 -3444.5

Importante:

- ▷ El literal 3 es un entero.
- ▷ El literal 3.0 es un real.

Los compiladores suelen usar el tipo `double` para representar literales reales.

También se puede utilizar *notación científica (scientific notation)* :

5.32e+5 representa el número $5.32 * 10^5 = 532000$

42.9e-2 representa el número $42.9 * 10^{-2} = 0.429$

I.5.2.2. Representación de los reales

¿Cómo podría el ordenador representar 541.341?

Lo *fácil* sería:

- ▷ Representar la parte entera 541 en binario
- ▷ Representar la parte real 341 en binario

De esa forma, con 64 bits (32 bits para cada parte) podríamos representar:

- ▷ Partes enteras en el rango $[-2147483648, 2147483647]$
- ▷ Partes reales en el rango $[-2147483648, 2147483647]$

Sin embargo, la forma usual de representación no es así. Se utiliza la representación en *coma flotante (floating point)*. La idea es representar un *valor* y la *escala*. En aritmética decimal, la escala se mide con potencias de 10:

$$42.001 \rightarrow \text{valor} = 4.2001 \quad \text{escala} = 10$$

$$42001 \rightarrow \text{valor} = 4.2001 \quad \text{escala} = 10^4$$

$$0.42001 \rightarrow \text{valor} = 4.2001 \quad \text{escala} = 10^{-1}$$

El valor se denomina *mantisa (mantissa)* y el coeficiente de la escala *exponente (exponent)*.

En la representación en coma flotante, la base de la escala es 2. A *grosso modo* se utilizan m bits para la mantisa y n bits para el exponente. La forma explícita de representación en binario se verá en otras asignaturas. Basta saber que utiliza potencias inversas de 2. Por ejemplo, **1011** representaría

$$1 * \frac{1}{2^1} + 0 * \frac{1}{2^2} + 1 * \frac{1}{2^3} + 1 * \frac{1}{2^4} =$$

$$= 1 * \frac{1}{2} + 0 * \frac{1}{4} + 1 * \frac{1}{8} + 1 * \frac{1}{16} = 0.6875$$

Problema: Si bien un entero se puede representar de forma exacta como suma de potencias de dos, un real sólo se puede *aproximar* con suma de potencias inversas de dos.

Valores tan sencillos como 0.1 o 0.01 no se pueden representar de forma exacta, produciéndose un error de *redondeo (rounding)*

$$0.1 \approx 1 * \frac{1}{2^4} + 0 * \frac{1}{2^5} + 0 * \frac{1}{2^6} + 0 * \frac{1}{2^7} + 0 * \frac{1}{2^8} + \dots$$

Por tanto:

Todas las operaciones realizadas con datos de tipo real pueden devolver valores que sólo sean aproximados

IMPORTANT

Especial cuidado tendremos con operaciones del tipo

Repita varias veces

Ir sumándole a una `variable_real` varios valores reales;

ya que los errores de aproximación se irán acumulando.



Ampliación:



En aplicaciones gráficas, por ejemplo, el uso de datos de coma flotante es usual y las GPU están optimizadas para realizar operaciones con ellos.

Sin embargo, en otras aplicaciones, como por ejemplo las bancarias, el uso de datos en coma flotante no está recomendado (incluso prohibido por la UE) En su lugar, se usan estándares (implementados en clases específicas) como por ejemplo BCD [Decimal codificado en binario \(Binary-coded decimal\)](#) , en el que cada cifra del número se representa por separado (se necesitan 4 bits para cada cifra) Esto permite representar con exactitud cualquier número decimal, aunque las operaciones a realizar con ellos son más complicadas de implementar y más lentas de ejecutar.

I.5.2.3. Rango y Precisión

La codificación en coma flotante separa el valor de la escala. Esto permite trabajar (en un mismo tipo de dato) con magnitudes muy grandes y muy pequeñas.

```
double masa_tierra_kg, masa_electron_kg;

masa_tierra_kg = 5.98e24;           // ok
masa_electron_kg = 9.11e-31;       // ok
```

C++ ofrece varios tipos reales: float, double, long double, etc. Con 64 bits, pueden representarse exponentes hasta ± 308 .

Pero el precio a pagar es muy elevado ya que se obtiene muy poca **precisión (precision)** (número de dígitos consecutivos que pueden representarse) tanto en la parte entera como en la parte real.

Tipo	Tamaño	Rango	Precisión
float	4 bytes	+/-3.4 e +/-38	7 dígitos aproximadamente
double	8 bytes	+/-1.7 e +/-308	15 dígitos aproximadamente

```
double valor_real;
```

```
// Datos de tipo double con menos de 15 cifras  
// (en su representación decimal)
```

```
valor_real = 11.0;
```

```
// Almacena: 11.0
```



Correcto

```
valor_real = 1.1;
```

```
// Almacena: 1.1000000000000001
```



Problema de redondeo

```
// Datos de tipo double con más de 15 cifras  
// (en su representación decimal)
```

```
valor_real = 1000000000000000000100.0;
```

```
// Almacena: 1000000000000000000000.0
```



Problema de precisión

```
valor_real = 0.100000000000000000009;
```

```
// Almacena: 0.10000000000000001
```



Problema de precisión

```
valor_real = 1.000000000000000000009;
```

```
// Almacena: 1.0
```



Problema de precisión

```
valor_real = 10000000001.000000000004;
```

```
// Almacena: 10000000001.0
```



Problema de precisión

En resumen:

- ▷ Los tipos **enteros** representan datos enteros de forma exacta, siempre que el valor esté en el rango correspondiente.
- ▷ Los tipos **reales** representan la **parte entera** de forma exacta si el número de dígitos es menor o igual que 7 -16 bits- o 15 -32 bits-. La representación es aproximada si el número de dígitos es mayor. La **parte real** será sólo aproximada.

I.5.2.4. Indeterminación e Infinito

Los reales en coma flotante también permiten representar valores especiales como **infinito** (*infinity*) y una **indeterminación** (*undefined*) (*Not a Number*)

En C++11, hay sendas constantes (realmente cada una es una **macro**) llamadas `INFINITY` y `NAN` para representar ambos valores. Están definidas en `cmath`.

Las operaciones numéricas con infinito son las usuales en Matemáticas (1.0/`INFINITY` es cero, por ejemplo) mientras que cualquier expresión que involucre `NAN`, produce otro `NAN`:

```
double valor_real, divisor = 0.0;

valor_real = 17.5 / divisor;           // Almacena INFINITY
valor_real = 1.5 / valor_real;         // Almacena 0.0
valor_real = divisor / divisor;        // Almacena NAN
valor_real = 1.5 / valor_real;         // Almacena NAN
valor_real = 1e+300;
valor_real = valor_real * valor_real;  // Almacena INFINITY
valor_real = 1.0 / valor_real;         // Almacena 0.0
```

I.5.2.5. Operadores

Los operadores matemáticos usuales también se aplican sobre datos reales:

+, -, *, /

Binarios, de notación infija. También se puede usar el operador unario de cambio de signo (-). Aplicados sobre reales, devuelven un real.

```
double real;  
real = 5.0 * 7.0;    // Asigna a real el valor 35.0  
real = 5.0 / 7.0;    // Asigna a real el valor 0.7142857
```

¡Cuidado! El comportamiento del operador / depende del tipo de los operandos: si todos son enteros, es la división entera. Si todos son reales, es la división real.

```
5 / 7      es una expresión entera. Resultado = 0  
5.0 / 7.0  es una expresión real. Resultado = 0.7142857
```

Si un argumento es entero y el otro real, la división es real.

```
5 / 7.0    es una expresión real. Resultado = 0.7142857
```

El operador módulo % no se puede aplicar sobre reales (no tienen sentido hacer la división entera entre reales). Su uso produce un error en compilación.

I.5.2.6. Funciones estándar

Hay algunas bibliotecas *estándar* que proporcionan funciones que trabajan sobre datos numéricos (enteros o reales) y que suelen devolver un real. Por ejemplo, `cmath` contiene, entre otras, las siguientes funciones:

```
pow(), cos(), sin(), sqrt(), tan(), log(), log10(), ....
```

Todas estas funciones son unarias excepto `pow`, que es binaria (base, exponente). Devuelven un real.

Para calcular el valor absoluto se usa la función `abs()`. Devuelve un tipo real (aún cuando el argumento sea entero).

```
#include<iostream>
#include <cmath>

using namespace std;

int main(){
    double real, otro_real;

    real      = 5.4;
    otro_real = abs(-5.4);
    otro_real = abs(-5);
    otro_real = sqrt(real);
    otro_real = pow(4.3, real);
}
```

Nota:

Observe que una misma función (`abs` por ejemplo) puede trabajar con datos de distinto tipo. Esto es posible porque hay varias sobrecargas de esta función. Posteriormente se verá con más detalle este concepto.

I.5.2.7. Expresiones reales

Son expresiones cuyo resultado es un número real.

`sqrt(real)` es una expresión real

`pow(4.3, real)` es una expresión real

Precedencia de operadores en las expresiones reales:

()
- (operador unario de cambio de signo)
* /
+ -

Consejo: *Para facilitar la lectura de las fórmulas matemáticas, evite el uso de paréntesis cuando esté claro cuál es la precedencia de cada operador.*



Ejemplo. Construya una expresión para calcular la siguiente fórmula:

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

<code>-b+sqrt(b*b-4.0*a*c)/2.0*a</code>	Error lógico
<code>((-b)+(sqrt((b*b) - (4.0*a)*c)))/(2.0*a)</code>	Difícil de leer
<code>(-b + sqrt(b*b - 4.0*a*c)) / (2.0*a)</code>	Correcto

Ejercicio. Construya una expresión para calcular la distancia euclídea entre dos puntos del plano $P1 = (x_1, y_1)$, $P2 = (x_2, y_2)$. Use las funciones `sqrt` y `pow`.

$$d(P1, P2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

I.5.3. Operando con tipos numéricos distintos

En este apartado vamos a ver que es posible mezclar datos numéricos de distinto tipo en una misma expresión. Al final, la expresión devolverá un valor numérico. En general, diremos que las *expresiones aritméticas* (*arithmetic expression*) o *numéricas* son las expresiones que devuelven un valor numérico, es decir, un entero o un real.

I.5.3.1. Asignaciones entre datos de distinto tipo

Vamos a analizar la siguiente situación:

```
dato_del_tipo_A = dato_del_tipo_B;
```

El operador de asignación permite trabajar con tipos distintos en la parte izquierda y derecha.

Si el tipo del resultado obtenido en la parte derecha de la asignación es distinto al del dato de la parte izquierda, el compilador realiza una *transformación de tipo automática* (*implicit conversion*) de la expresión de la derecha al tipo de dato de la parte izquierda de la asignación. También se conoce con el nombre de *casting*.

Esta transformación es temporal (mientras se evalúa la expresión)

- ▷ En general, a un dato numérico de tipo *pequeño* se le puede asignar cualquier expresión numérica de tipo *grande*. Si el resultado está en el rango permitido del tipo pequeño, la asignación se realiza correctamente. En otro caso, se produce un desbordamiento aritmético y se almacenará un valor indeterminado.

```
int chico;  
long long grande;
```

```
// chico = grande; Puede desbordarse.
```

```
grande = 6000000; // 6000000 es un literal int  
                // 6000000 int -> 6000000 long long  
                // long long = long long
```

```
chico = grande; // grande long long -> grande int int  
               // El resultado 6000000 cabe en chico  
               // int = int
```



```
grande = 3600000000000000;
```

```
                // 3600000000000000 es un literal long long  
                // long long = long long
```



```
chico = grande; // 3600000000000000 no cabe en un int  
               // Desbordamiento en el casting automático.  
               // chico = -415875072  
               // int = int
```



El literal 3600000000000000 **cabe en un long long pero no en un int**. Por lo tanto, en la asignación `chico = grande`, se produce un error lógico denominado **desbordamiento aritmético (arithmetic overflow)** y el resultado de asignar 3600000000000000 a un `int` es un valor indeterminado. En este caso, el valor concreto es -415875072.

Hay que destacar que no se produce ningún error de compilación ni durante la ejecución. Son errores difíciles de detectar con los que habrá que tener especial cuidado.

- ▷ **A un entero se le puede asignar una expresión real. En este caso, se pierde la parte decimal, es decir, se *trunca* la expresión real.**

```
double real;  
int entero;  
  
real = 5.3;           // 5.3 es un literal double  
                      // double = double  
  
// entero = real; Se trunca el real  
  
entero = real;        // real double (5.3) -> real int (5)  
                      // int = int
```

En resumen:

A un dato numérico se le puede asignar una expresión de un tipo distinto. Si el resultado cabe, no hay problema. En otro caso, se produce un desbordamiento aritmético y se asigna un valor indeterminado. Es un error lógico, pero no se produce un error de ejecución.

Si asignamos una expresión real a un entero, se trunca la parte decimal.

Ampliación:



Para obtener directamente los límites de los rangos de cada tipo, puede usarse `limits`:

```
#include <iostream>
#include <limits>          // -> numeric_limits

using namespace std;

int main(){
    // Rangos de int y double:

    cout << "Rangos de int y double:\n";
    cout << "int:\n";
    cout << numeric_limits<int>::min() << "\n";           // -2147483648
    cout << numeric_limits<int>::max() << "\n";           // 2147483647

    cout << "double:\n";
    cout << numeric_limits<double>::min() << "\n";        // 2.22507e-308
    cout << numeric_limits<double>::max() << "\n";        // 1.79769e+308
    cout << numeric_limits<double>::lowest() << "\n";     // -1.79769e-308
}
```

I.5.3.2. Asignaciones a datos de expresiones del mismo tipo

Vamos a analizar la siguiente situación:

```
variable_del_tipo_A = expresión_con_datos_del_tipo_A;
```

Si en una expresión todos los datos son del mismo tipo, la expresión devuelve un dato de ese tipo.

► Casos no problemáticos

Ejemplo. Suma de dos enteros pequeños

```
int chico;
```

```
chico = 2;
```

```
chico = chico + 1;    // int + int -> int. Sin problemas.
```



chico es de tipo de dato int, al igual que el literal 1. Por tanto, la expresión chico + 1 es de tipo de dato int. Todo funciona como cabría esperar.

Ejemplo. Calcular la longitud de una circunferencia

```
const double PI = 3.1415927;
```

```
double radio;
```

```
.....
```

```
longitud = 2.0 * PI * radio;    // double * double * double -> double
```

```
// Sin problemas.
```



En la evaluación de 2.0 * PI * radio intervienen dos variables de tipo double y un literal (2.0) también de tipo double. Como todos los datos son del mismo tipo, la expresión es de ese tipo (en esta caso, double) El resultado de evaluar la expresión 2.0 * PI * radio se asigna a longitud.

Ampliación:



Hemos dicho que si todos los datos de una expresión son de un mismo tipo de dato, la expresión devuelve un dato de dicho tipo. La excepción a esto es que las operaciones aritméticas con datos de tipo `char` devuelven un tipo `int`. Así pues, la expresión

`'B' - 'A'`

es una expresión de tipo `int` y no de tipo `char`, aunque todos los datos sean de ese tipo.

► Casos problemáticos

Ejemplo. Expresión fuera de rango

```
int chico;
```

```
chico = 2147483647;    // máximo int. Sin problemas.
```

```
chico = chico + 1;    // máximo int representable + 1
```

```
// Desbordamiento. Resultado: -2147483648
```



Al ser `chico` y `1` de tipo `int`, la expresión `chico + 1` es de tipo de dato `int`. Por lo tanto, C++ no realiza ningún casting automático y alberga el resultado (2147483648) en un `int`. Como no cabe, se desborda dando como resultado -2147483648. Finalmente, se ejecuta la asignación.

Observe que el desbordamiento se ha producido durante la evaluación de la expresión y no en la asignación

Ejemplo. ¿Qué pasaría en este código?

```
int chico = 1234567890;
long long grande;

grande = chico * chico;

// grande = 304084036    Error lógico 😞
```

En la expresión `chico * chico` todos los datos son del mismo tipo (`int`). Por tanto no se produce casting y el resultado se almacena en un `int`. La multiplicación correcta es 1524157875019052100 pero no cabe en un `int`, por lo que se produce un desbordamiento aritmético y a la variable `grande` se le asigna un valor indeterminado (304084036)

Observe que el resultado (1524157875019052100) sí cabe en un `long long` pero el desbordamiento se produce durante la evaluación de la expresión, **antes** de realizar la asignación.

Posibles soluciones: Lo veremos en la página 75

Nota:

El desbordamiento como tal no ocurre con los reales ya que una operación que de un resultado fuera de rango devuelve infinito (INF)

```
double real, otro_real;

real = 1e+200;
otro_real = real * real;
// otro_real = INF
```

I.5.3.3. Expresiones con datos numéricos de distinto tipo

Vamos a analizar la siguiente situación:

```
variable_de_cualquier_tipo = expresión_con_datos_de_tipo_cualquiera;
```

Muchos operadores numéricos permiten que los argumentos sean expresiones de tipos distintos. Para evaluar el resultado de una expresión que contenga datos con tipos distintos, el compilador realiza un casting para que todos sean del mismo tipo y así poder hacer las operaciones.

Los datos de tipo pequeño se transformarán al mayor tipo de los otros datos que intervienen en la expresión.

Esta transformación es temporal (mientras se evalúa la expresión)

► *Casos no problemáticos*

Ejemplo. Calcular la longitud de una circunferencia

```
const double PI = 3.1415927;
double radio;
.....
longitud = 2 * PI * radio;    // double    = int * double * double
                             // int -> double
                             // double    = double * double * double
```

En la evaluación de `2 * PI * radio` intervienen dos variables de tipo `double` y un literal (2) de tipo `int`. Se produce el casting

```
2 int -> 2.0 double
```

Se evalúa la expresión `2.0 * PI * radio` y el resultado se asigna a `longitud`.

Ejemplo. Calcule la cuantía de las ventas totales de un producto a partir de su precio y del número de unidades vendidas.

```
int unidades_vendidas;
double precio_unidad, venta_total;
.....
cin >> precio_unidad;
cin >> unidades_vendidas;

venta_total = precio_unidad * unidades_vendidas;
    // double      * int
    // unidades_vendidas int -> double
    // double      * double
    // El resultado de la expresión es double
    // double = double
```



► Casos problemáticos

Ejemplo. Calcule la media aritmética de la edades de dos personas.

```
int edad1 = 10, edad2 = 5;  
double media;
```

```
media = (edad1 + edad2) / 2; // media = 7.0
```



Analicemos la situación:

- ▷ **Los datos** `edad1` **y** `edad2` **son** `int`, **por lo que la expresión** `edad1 + edad2` **es de tipo** `int` **y devuelve** 15.
- ▷ **2 es un literal** `int`.
- ▷ **Así pues, los dos operandos de la expresión** `(edad1 + edad2) / 2` **son enteros, por lo que el operador de división actúa sobre enteros y es la división entera, devolviendo el entero 7. Al asignarlo a la variable real** `media`, **se transforma en 7.0. Se ha producido un error lógico.**

Posibles soluciones (de peor a mejor)

- ▷ Usar un dato temporal de un tipo mayor. 😞

```
int edad1 = 10, edad2 = 5;
double media, edad1_double;

edad1_double = edad1;
media        = (edad1_double + edad2) / 2;
              // double + int es double
              // double / int es double
              // media = 7.5
```

El inconveniente de esta solución es que estamos representando un mismo dato con dos variables distintas y corremos el peligro de usarlas en sitios distintos (con valores diferentes).

Cada entidad que vaya a representar en un programa debe estar definida en un único sitio. Dicho de otra forma, no use varias variables para representar un mismo concepto.

IMPORTANT

- ▷ **Cambiar el tipo de dato original de las variables.** 😞

```
double edad1 = 10, edad2 = 5;
double media;

media = (edad1 + edad2) / 2;
// double + double es double
// double / int es double
// media = 7.5
```

Debemos evitar esta solución ya que el tipo de dato asociado a las variables debe depender de la semántica de éstas. En cualquier caso, sí sería lícito reconsiderar la elección de los tipos de los datos y cambiarlos si la semántica de éstos así lo demanda. No es el caso de nuestras variables de edad, que son enteras.

- ▷ **Usar un casting manual tal y como se indica en la sección [I.5.3.4](#) (página 78)** 😊
- ▷ **Forzamos la división real introduciendo un literal real:**

```
int edad1 = 10, edad2 = 5;
double media;

media = (edad1 + edad2) / 2.0; 😊
// int + int es int
// int / double es double
// media = 7.5
```

En resumen:

Durante la evaluación de una expresión numérica en la que intervienen datos de distinto tipo, el compilador realizará un casting para transformar los datos de tipos pequeños al mayor de los tipos involucrados.

Esta transformación es temporal (sólo se aplica mientras se evalúa la expresión).

Pero cuidado: si en una expresión todos los datos son del mismo tipo, el compilador no realiza ninguna transformación, de forma que la expresión resultante es del mismo tipo que la de los datos involucrados. Por tanto, cabe la posibilidad que se produzca un desbordamiento durante la evaluación de la expresión.

I.5.3.4. El operador de casting (Ampliación)

Este apartado es de ampliación. No entra en el examen.

El *operador de casting (casting operator)* permite que el programador pueda cambiar explícitamente el tipo por defecto de una expresión. La transformación es siempre temporal: sólo afecta a la instrucción en la que aparece el casting.

```
static_cast<tipo_de_dato> (expresión)
```

Ejemplo. Media aritmética:

```
int edad1 = 10, edad2 = 5;
double media;

media = (static_cast<double>(edad1) + edad2)/2;
```



Ejemplo. Retomamos el ejemplo de la página 71:

```
int chico = 1234567890;
long long grande;

grande = static_cast<long long>(chico) * chico;
// chico int -> chico long long
// long long * int
// grande = 1524157875019052100


// chico sigue siendo int después de la instrucción anterior
```



¿Qué ocurre en el siguiente ejemplo?

```
int chico = 1234567890;
long long grande;

grande = static_cast<long long> (chico * chico);

// grande = 304084036 
```

La expresión `chico * chico` es de tipo `int` ya que todos los datos que intervienen son de tipo `int`. Como un `int` no es capaz de almacenar el resultado de multiplicar 1234567890 consigo mismo, se produce un desbordamiento.

Nota:

En C, hay otro operador de casting que realiza una función análoga a `static_cast`:

(<tipo de dato>) expresión

```
int edad1 = 10, edad2 = 5;
double media;
media = ((double)edad1 + edad2)/2;
```

I.5.4. El tipo de dato cadena de caracteres

Un literal de tipo *cadena de caracteres (string)* es una sucesión de caracteres encerrados entre comillas dobles:

"Hola", "a" son literales de cadena de caracteres

```
cout << "Esto es un literal de cadena de caracteres";
```

Las secuencias de escape también pueden aparecer en los literales de cadena de caracteres.

```
int main(){  
    cout << "Bienvenidos";  
    cout << "\nEmpiezo a escribir en la siguiente línea";  
    cout << "\n\tAcabo de tabular esta línea";  
    cout << "\n";  
    cout << "\nEsto es una comilla simple '";  
    cout << " y esto es una comilla doble \"";  
}
```

Escribiría en pantalla:

Bienvenidos

Empiezo a escribir en la siguiente línea

Acabo de tabular esta línea

Esto es una comilla simple ' y esto es una comilla doble "

Nota:

Formalmente, el tipo `string` no es un tipo simple sino compuesto de varios caracteres. Lo incluimos dentro de este tema en aras de simplificar la materia.

Ejercicio. Determinar cuáles de las siguientes son constantes de cadena de caracteres válidas, y determinar la salida que tendría si se pasase como argumento a `cout`

- a) "8:15 P.M." b) "'8:15 P.M.'" c) '"8:15 P.M."'
- d) "Dirección\n" e) "Dirección'n" f) "Dirección\'n"
- g) "Dirección\\'n"

C++ ofrece dos alternativas para trabajar con cadenas de caracteres:

- ▷ **Cadenas estilo C:** son *vectores de caracteres* con terminador `'\0'`. Se verá en la asignatura Metodología de la Programación.
- ▷ **Usando el tipo `string`** (la recomendada en esta asignatura)

```
int main(){
    string mensaje_bienvenida;
    mensaje_bienvenida = "\tFundamentos de Programación\n";
    cout << mensaje_bienvenida;
}
```

Para poder operar con un `string` debemos incluir la biblioteca `string`:

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

int main(){
    string cad;
    cad = "Hola y ";
    cad = cad + "adiós";
    cout << cad;
}
```

Una función muy útil definida en la biblioteca `string` es:

```
to_string( <dato> )
```

donde `dato` puede ser casi cualquier tipo numérico. Para más información:

http://en.cppreference.com/w/cpp/string/basic_string/to_string

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

int main(){
    string cadena;
    int entero;
    double real;

    entero = 27;
    real    = 23.5;
    cadena = to_string(entero);    // Almacena "27"
    cadena = to_string(real);      // Almacena "23.500000"
}
```

Nota:

La función `to_string` es otro ejemplo de función que puede trabajar con argumentos de distinto tipo de dato como enteros, reales, etc (sobrecargas de la función)

También están disponibles funciones para hacer la transformación inversa, como por ejemplo:

```
stoi( <cadena> )      stod( <cadena> )
```

que convierten a `int` y `double` respectivamente:

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

int main(){
    string cadena;
    int entero;
    double real;

    cadena = "27";
    entero = stoi(cadena);          // Almacena 27
    cadena = "23.5";
    real   = stod(cadena);          // Almacena 23.5
    cadena = " 23.5 basura";
    real   = stod(cadena);          // Almacena 23.5
    cadena = "basura 23.5";
    real   = stod(cadena);          // Error de ejecución
}
```

Ampliación:

Realmente, `string` es una clase (lo veremos en temas posteriores) por lo que le serán aplicables métodos en la forma:

```
cad = "Hola y ";
cad.append("adiós");    // :-0
```



La *cadena vacía (empty string)* es un literal especial de cadena de caracteres. Se especifica a través del token `""`:

```
string cadena;
```

```
cadena = "";
```

El tipo de dato `string` tiene la particularidad de que todos los datos de dicho tipo, son inicializados por C++ a la cadena vacía:

```
string cadena; // Contiene ""  
               // No contiene un valor indeterminado
```

En temas posteriores veremos con más detalle la manipulación de cadenas de caracteres.

I.5.5. El tipo de dato carácter

I.5.5.1. Representación de caracteres en el ordenador

Este apartado I.5.5.1 es de introducción. No hay que memorizar nada.

Frecuentemente, queremos manejar información que podemos representar con un único carácter. Por ejemplo, grupo de teoría de una asignatura, carácter a leer desde el teclado para seleccionar una opción de un menú, calificación obtenida (según la escala ECTS), etc. Pueden ser tan básicos como las letras del alfabeto inglés, algo más particulares como las letras del alfabeto español o complejos como los jeroglifos egipcios.

Cada plataforma (ordenador, sistema operativo, interfaz de usuario, etc) permitirá el uso de cierto *conjunto de caracteres (character set)* . Para su manejo, se establece una enumeración, asignando un *número de orden (code point)* a cada carácter, obteniéndose una tabla o *página de códigos (code page)* . La tabla más antigua es la tabla *ASCII* . Se compone de 128 caracteres. Los primeros 31 son caracteres de control (como por ejemplo fin de fichero) y el resto son caracteres imprimibles:

Code	Char	Code	Char	Code	Char	Code	Char	Code	Char	Code	Char
32	[space]	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	\$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(56	8	72	H	88	X	104	h	120	x
41)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	
45	-	61	=	77	M	93]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	[backspace]

Cualquier página de códigos debe ser un super-conjunto de la tabla ASCII. Por ejemplo, la página de códigos denominada **ISO/IEC 8859-1** permite representar la mayor parte de los caracteres usados en la Europa Occidental (conjunto de caracteres que suele denominarse *Latin-1*) En esta tabla, la ñ, por ejemplo, ocupa la posición 241:

0	32	64 @	96 ‘	128	160	192 À	224 à
1	33 !	65 A	97 a	129	161 ÿ	193 Á	225 á
2	34 "	66 B	98 b	130	162 ø	194 Â	226 â
3	35 #	67 C	99 c	131	163 £	195 Ã	227 ã
4	36 \$	68 D	100 d	132	164 ¢	196 Ä	228 ä
5	37 %	69 E	101 e	133	165 ¥	197 Å	229 å
6	38 &	70 F	102 f	134	166 †	198 Æ	230 æ
7	39 ’	71 G	103 g	135	167 §	199 Ç	231 ç
8	40 (72 H	104 h	136	168 ¨	200 È	232 è
9	41)	73 I	105 i	137	169 ©	201 É	233 é
10	42 *	74 J	106 j	138	170 ®	202 Ê	234 ê
11	43 +	75 K	107 k	139	171 «	203 Ë	235 ë
12	44 ,	76 L	108 l	140	172 ¬	204 Ì	236 ì
13	45 -	77 M	109 m	141	173 -	205 Í	237 í
14	46 .	78 N	110 n	142	174 ®	206 Î	238 î
15	47 /	79 O	111 o	143	175 -	207 Ï	239 ï
16	48 0	80 P	112 p	144	176 °	208 Ð	240 ð
17	49 1	81 Q	113 q	145	177 ±	209 Ñ	241 ñ
18	50 2	82 R	114 r	146	178 ²	210 Ò	242 ò
19	51 3	83 S	115 s	147	179 º	211 Ó	243 ó
20	52 4	84 T	116 t	148	180 ´	212 Ô	244 ô
21	53 5	85 U	117 u	149	181 µ	213 Õ	245 õ
22	54 6	86 V	118 v	150	182 ¶	214 Ö	246 ö
23	55 7	87 W	119 w	151	183 ·	215 ×	247 ÷
24	56 8	88 X	120 x	152	184 ,	216 Ø	248 ø
25	57 9	89 Y	121 y	153	185 ´	217 Ù	249 ù
26	58 :	90 Z	122 z	154	186 º	218 Ú	250 ú
27	59 ;	91 [123 {	155	187 »	219 Û	251 û
28	60 <	92 \	124	156	188 ¼	220 Ü	252 ü
29	61 =	93]	125 }	157	189 ½	221 Ý	253 ý
30	62 >	94 ^	126 ~	158	190 ¾	222 Þ	254 þ
31	63 ?	95 _	127	159	191 ¿	223 ß	255 ÿ

Otra página muy usada en Windows es Windows-1252 que es una ligera

modificación de la anterior.

Si bien los primeros 128 caracteres de todas las páginas son los mismos (la tabla básica ASCII), el resto no tienen por qué serlo. Esto ocasiona muchos problemas de portabilidad.

Para aumentar el problema, también es distinta la **codificación (coding)** usada, es decir, la forma en la que cada plataforma representa en memoria cada uno de dichos caracteres (realmente los code points), pues pueden utilizarse dos bytes, cuatro bytes, un número variable de bytes, etc.

Para resolver este problema, se diseñó el estándar **ISO-10646** más conocido por **Unicode** (multi-lenguaje, multi-plataforma). Este estándar es algo más que una página de códigos ya que no sólo establece el conjunto de caracteres que pueden representarse (incluye más de un millón de caracteres de idiomas como español, chino, árabe, jeroglifos, etc), sino también especifica cuál puede ser su codificación (permite tres tamaños distintos: 8, 16 y 32 bits) y asigna un número de orden o **code point** a cada uno de ellos. Los caracteres de ISO-8859-1 son los primeros 256 caracteres de la tabla del estándar Unicode.

Nota:

Para mostrar en Windows el carácter asociado a un code point con orden en decimal x, hay que pulsar ALT 0x. Por ejemplo, para mostrar el carácter asociado al codepoint con orden 35, hay que pulsar ALT 035

Para rematar los problemas, hay que tener en cuenta que la codificación usada en el editor de texto de nuestro programa en C++ podría ser distinta a la usada en la consola de salida de resultados.

Por lo tanto, para simplificar, en lo que sigue supondremos que tanto la plataforma en la que se está escribiendo el código fuente, como la consola que se muestra al ejecutar el programa usa un súper conjunto de la tabla definida en ISO-8859-1.

Para poder mostrar en la consola de MSDOS caracteres de ISO-8859-1 como la ñ o las letras acentuadas, hay que realizar ciertos cambios. En el guión de prácticas se indican con detalle los pasos a dar.

I.5.5.2. Literales de carácter

Son tokens formados por:

- ▷ O bien un único carácter encerrado entre comillas simples:

'!' 'A' 'a' 'ñ' '5'

Observe lo siguiente:

'5' es un literal de carácter

5 es un literal entero

"5" es un literal de cadena de caracteres

'cinco' no es un literal de carácter correcto

'11' no es un literal de carácter correcto

- ▷ O bien una *secuencia de escape* entre comillas simples. Dichas secuencias se forman con el símbolo \ seguido de otro símbolo, como por ejemplo:

Secuencia	Significado
'\n'	Nueva línea (retorno e inicio)
'\t'	Tabulador
'\b'	Retrocede 1 carácter
'\r'	Retorno de carro
'\f'	Salto de página
'\"'	Comilla simple
'\"'	Comilla doble
'\\'	Barra inclinada

Ejemplo.

```
cout << 'c' << 'a' << 'ñ' << 'a';  
cout << '\n' << '!' << '\n' << 'B' << '1';  
cout << '\n' << '\t' << '\"' << 'B' << 'y' << 'e' << '\"';
```

Salida en pantalla:

```
caña  
!  
B1  
    "Bye"
```

Las secuencias de escape también pueden ir dentro de un literal de cadena de caracteres, por lo que, realmente, hubiese sido más cómodo sustituir el anterior código por éste:

```
cout << "caña";  
cout << "\n!\nB1";  
cout << "\n\t\"Bye\"";
```

I.5.5.3. Asignación de literales de carácter

Ya sabemos imprimir literales de carácter con `cout`. Ahora bien, ¿podemos asignar un literal de carácter a un dato variable o constante?

- ▷ Podemos asignar un literal de carácter a un dato de tipo cadena de caracteres.
- ▷ Podemos asignar un literal de carácter a cualquier dato de tipo entero.

Con las cadenas de caracteres, todo funciona como cabría esperar:

```
string cadena_letra_piso;  
  
cadena_letra_piso = "A"; // Almacena "A"  
cadena_letra_piso = 'A'; // Almacena "A"
```

Con los tipos enteros, el valor que el entero almacena es el número de orden del carácter en la tabla:

```
int letra_piso;  
long entero_grande;  
  
letra_piso    = 65; // Almacena 65  
letra_piso    = 'A'; // Almacena 65  
entero_grande = 65; // Almacena 65  
entero_grande = 'A'; // Almacena 65
```

Ya sabemos que lo siguiente no es correcto:

```
int letra_piso;

letra_piso = "A"; // Error de compilación. Es un literal de cadena
letra_piso = A;   // Error de compilación. Faltan las comillas
                  // Si A fuese una variable int, sí sería correcto
letra_piso = 'A'; // Error de compilación. No son esas comillas

int letra_piso;

letra_piso = 241;      // Almacena 241
letra_piso = 'ñ';      // Almacena 241
```

I.5.5.4. El tipo de dato `char`

Trabajar con literales de carácter y un tipo de dato entero tiene algunos problemas:

- ▷ Si sólo necesitamos manejar los 128 caracteres de la tabla ASCII (o como mucho los 256 de cualquier tabla similar a ISO-8859-1) un tipo entero utiliza bastante más memoria de la necesaria.
- ▷ Los operadores de E/S `cin` y `cout` realizan la salida o entrada atendiendo al tipo de dato.

```
int letra_piso;

cin  >> letra_piso;    // Al introducir A se produce un error
letra_piso = 'A';      // Almacena 65
cout << letra_piso;    // Imprime 65. No imprime A
letra_piso = 65;       // Almacena 65
cout << letra_piso;    // Imprime 65. No imprime A
```

Para resolver estos problemas, C++ introduce otros tipos de datos:

- ▷ Siguen siendo tipos enteros, normalmente más pequeños.
- ▷ Las operaciones de E/S están diseñadas para trabajar con caracteres.

Nosotros sólo veremos el tipo de dato `char` ya que es el más usado en C++ para trabajar con caracteres: (pero hay otros como `wchar_t`, `char32_t`, etc)

- ▷ El tipo `char` es un tipo entero pequeño pero suficiente como para albergar al menos 256 valores distintos (así se indica en C++ 14) por lo que, por ejemplo, podemos trabajar con los caracteres de la página de códigos ISO-8859-1
- ▷ Captura e imprime caracteres correctamente cuando se utiliza con `cin` y `cout` respectivamente.

```
char letra_piso;

letra_piso = 65;      // Almacena 65
letra_piso = 'A';     // Almacena 65

cout << letra;        // Imprime A
cin  >> letra;        // Usuario introduce B
                        // Almacena 66

cout << letra;        // Imprime B
cin  >> letra;        // Usuario introduce ñ
                        // Almacena 241

cout << letra;        // Imprime ñ
```

El tipo de dato `char` es un entero usualmente pequeño que permite albergar al menos 256 valores distintos (así se indica en C++ 14) y está pensado para realizar operaciones de E/S con caracteres.

Ejemplo.

```
#include <iostream>
using namespace std;
int main(){
    const char NUEVA_LINEA = '\n';
    char letra_piso;

    letra_piso = 'B';    // Almacena 66 ('B')
    cout << letra_piso << "ienvenidos";
    cout << '\n' << "Empiezo a escribir en la siguiente línea";
    cout << '\n' << '\t' << "Acabo de tabular esta línea";
    cout << NUEVA_LINEA;
    cout << '\n' << "Esto es una comilla simple: " << '\'';
}
```

Escribiría en pantalla:

```
Bienvenidos
Empiezo a escribir en la siguiente línea
    Acabo de tabular esta línea

Esto es una comilla simple: '
```

Ampliación:

Para escribir un retorno de carro, también puede usarse una constante llamada `endl` en la forma:

```
cout << endl << "Adiós" << endl
```

Esta constante, además, obliga a vaciar el buffer de datos en ese mismo momento, algo que, por eficiencia, no siempre queremos hacer.



Ampliación:



El estándar de C++ sólo impone que el tamaño para un tipo `char` debe ser menor o igual que el resto de los enteros. Algunos compiladores usan un tipo de 1 byte y capacidad de representar sólo positivos (0..255) y otros compiladores representan en un `char` números negativos y positivos (-127..127) (eso sin tener en cuenta si la representación interna es como complemento a 2 o no)

Así pues, tenemos doble lío:

- ▷ La página de códigos con los que cada plataforma trabaja es distinto.
- ▷ El tipo de dato `char` de C++ no tiene una misma representación entre los distintos compiladores.

Es por ello que a lo largo de la asignatura asumiremos que el compilador cumple el estándar C++14 y por tanto, el tipo `char` permite almacenar 256 valores distintos. Además supondremos que la plataforma con la que trabajamos utiliza la página de códigos ISO-8859-1, por lo que los 256 caracteres de esta tabla pueden ser mostrados en pantalla y representados en un `char`

Ampliación:



Cabría esperar que C++ proporcionase una forma fácil de trabajar con caracteres Unicode, pero no es así. Si bien C++11 proporciona el tipo `char32_t` para almacenar caracteres Unicode, su manipulación debe hacerse con librerías específicas.

I.5.5.5. Funciones estándar y operadores

El fichero de cabecera `cctype` contiene varias funciones para trabajar con caracteres. Los argumentos y el resultado son de tipo `int`. Por ejemplo:

```
                                tolower  toupper

#include <cctype>
using namespace std;

int main(){
    char letra_piso;    // También valdría cualquier tipo entero

    letra_piso = tolower('A');    // Almacena 97 ('a')
    letra_piso = toupper('A');    // Almacena 65 ('A')
    letra_piso = tolower('B');    // Almacena 98 ('b')
    letra_piso = tolower('!');    // Almacena 33 ('!') No cambia
```

Los operadores aplicables a los enteros también son aplicables a cualquier `char` o a cualquier literal de carácter. El operador actúa siempre sobre el valor de orden que le corresponde en la tabla ASCII (la página de códigos en general)

```
char character;    // También valdría cualquier tipo entero
int diferencia;

character = 'A' + 1;    // Almacena 66 ('B')
character = 65 + 1;    // Almacena 66 ('B')
character = '7' - 1;    // Almacena 54 ('6')

diferencia = 'c' - 'a';    // Almacena 2
```

Ejercicio. ¿Qué imprimiría la sentencia `cout << 'A'+1`? No imprime 'B' como cabría esperar sino 66. ¿Por qué?

I.5.6. Lectura de varios datos

Hasta ahora hemos leído/escrito datos uno a uno desde/hacia la consola, separando los datos con `Enter`.

```
int entero, otro_entero;
double real;

cin >> entero;
cin >> real;
cin >> otro_entero;
```

Si se desea, pueden leerse los valores en la misma línea:

```
cin >> entero >> real >> otro_entero;
```

En cualquiera de los dos casos, cuando procedamos a introducir los datos, también podríamos haber usado como separador un espacio en blanco o un tabulador. ¿Cómo funciona?

La E/S utiliza un *buffer* intermedio. Es un espacio de memoria que sirve para ir suministrando datos para las operaciones de E/S, desde el dispositivo de E/S al programa. Por ahora, dicho dispositivo será el teclado.

El primer `cin` pide datos al teclado. El usuario los introduce y cuando pulsa `Enter`, éstos pasan al buffer y termina la ejecución de `cin >> entero;`. Todo lo que se haya escrito en la consola pasa al buffer, *incluido* el `Enter`. Éste se almacena como el carácter `'\n'`.

Sobre el buffer hay definido un *cursor (cursor)* que es un apuntador al siguiente byte sobre el que se va a hacer la lectura. Lo representamos con `↑`. Representamos el espacio en blanco con `␣`

Supongamos que el usuario introduce 43 52.1<Enter>

43	□	□	52.1	\n		cin >> entero;		□	□	52.1	\n
↑						entero = 43		↑			

El 43 se asigna a `entero` y éste se borra del buffer.

Las ejecuciones posteriores de `cin` se saltan, previamente, los separadores que hubiese al principio del buffer (espacios en blanco, tabuladores y `\n`). Dichos separadores se eliminan del buffer.

La lectura se realiza sobre los datos que hay en el buffer. Si no hay más datos en él, el programa los pide a la consola.

Ejemplo. Supongamos que el usuario introduce 43 52.1<Enter>

```
cin >> entero;
// Usuario:  43      52.1<Enter>
// Buffer:    [43     52.1\n]
// entero =   43
// Buffer:    [      52.1\n]
cin >> real;
// real = 52.1
// Buffer:    [\n]
cin >> otro_entero;
// Buffer:    []
```

Ahora el buffer está vacío, por lo que el programa pide datos a la consola:

```
// Usuario: 37<Enter>
// otro_entero = 37;
// Buffer:    [\n]
```

Esta comunicación funciona igual entre un fichero y el buffer. Para que la entrada de datos sea con un fichero en vez de la consola basta ejecutar el programa desde el sistema operativo, redirigiendo la entrada:

```
C:\mi_programa.exe < fichero.txt
```

Contenido de fichero.txt:

```
43      52.1\n37
```

Desde un editor de texto se vería lo siguiente:

```
43      52.1
37
```

La lectura sería así:

```
cin >> entero;
    // Buffer:  [43      52.1\n37]
    // entero =  43
    // Buffer:  [      52.1\n37]
cin >> real;
    // real = 52.1
    // Buffer:  [\n37]
cin >> otro_entero;
    // otro_entero = 37;
    // Buffer:  []
```

¿Qué pasa si queremos leer un entero pero introducimos, por ejemplo, una letra?

- ▷ Si la letra se introduce después del número, no se produce ningún error.

□ 1 2 3 a □	<code>cin >> entero;</code>	
↑	<code>entero = 123</code>	a □
	<code>cin >> caracter;</code>	↑
	<code>caracter = 'a'</code>	

- ▷ Si se lee un entero y lo primero que se introduce es una letra, se produce un error en la lectura y a partir de ese momento, todas las operaciones siguientes de lectura también dan fallo y el cursor no avanzaría.

□ □ a □ 1 2 3	<code>cin >> entero;</code>	
↑	Fallo de lectura	□ □ a □ 1 2 3
	<code>cin >> lo_que_sea;</code>	↑
	Fallo de lectura	

Se puede resetear el estado de la lectura con `cin.clear()` y consultarse el estado actual (error o correcto) con `cin.fail()`. En cualquier caso, para simplificar, a lo largo de este curso asumiremos que los datos vienen en el orden correcto especificado en el programa, por lo que no será necesario recurrir a `cin.clear()` ni a `cin.fail()`.

Si vamos a leer sobre un tipo `char` debemos tener en cuenta que `cin` siempre se salta los separadores y saltos de línea que previamente hubiese:

```
char character;
```

<code>\n \n a 1 2 3</code> \uparrow	<code>cin >> character;</code> <code>character = 'a'</code>	<code>1 2 3</code> \uparrow
--	--	----------------------------------

Si queremos leer los separadores (por ejemplo, el espacio en blanco) en una variable de tipo `char` debemos usar `cin.get()`:

<code>a 1 2 3</code> \uparrow	<code>character = cin.get();</code> <code>character = ' '</code>	<code>a 1 2 3</code> \uparrow
<code>a 1 2 3</code> \uparrow	<code>character = cin.get();</code> <code>character = 'a'</code>	<code>1 2 3</code> \uparrow

Lo mismo ocurre si hubiese un carácter de nueva línea:

<code>\n \n a \n</code> \uparrow	<code>character = cin.get();</code> <code>character = '\n'</code>	<code>\n a \n</code> \uparrow
---------------------------------------	--	------------------------------------

Ampliación:

Para leer una cadena de caracteres (`string`) podemos usar la función `getline` de la biblioteca `string`. Permite leer caracteres hasta llegar a un terminador que, por defecto, es el carácter de nueva línea `'\n'`. La cadena a leer se pasa como un parámetro por referencia a la función. Este tipo de parámetros se estudian en el segundo cuatrimestre.



I.5.7. El tipo de dato lógico o booleano

Es un tipo de dato muy común en los lenguajes de programación. Se utiliza para representar los valores verdadero y falso que suelen estar asociados a una condición.

En C++ se usa el tipo `bool`.

I.5.7.1. Rango

El rango de un dato de tipo *lógico (boolean)* está formado solamente por dos valores: verdadero y falso. Para representarlos, se usan los siguientes literales:

`true` `false`

I.5.7.2. Funciones estándar y operadores lógicos

Una expresión lógica es una expresión cuyo resultado es un tipo de dato lógico.

Algunas funciones que devuelven un valor lógico:

▷ **En la biblioteca cctype:**

```
isalpha    isalnum    isdigit    ...
```

Por ejemplo, `isalpha('3')` es una expresión lógica (devuelve `false`)

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

int main(){
    bool es_alfabetico, es_alfanumerico, es_digito_numerico;

    es_alfabetico      = isalpha('3');    // false
    es_alfanumerico    = isalnum('3');    // true
    es_digito_numerico = isdigit('3');    // true
```

▷ **En la biblioteca cmath:**

```
isnan      isinf      isfinite    ...
```

Comprueban, respectivamente, si un real contiene `NAN`, `INFINITY` o si es distinto de los dos anteriores.

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

int main(){
    double real = 0.0;
    bool es_indeterminado;

    real = real/real;
    es_indeterminado = isnan(real);    // true
```

Los operadores son los clásicos de la lógica Y, O, NO que en C++ son los operadores `&&`, `||`, `!` respectivamente.

$p \equiv$ Carlos es varón

$q \equiv$ Carlos es joven

p	q	$p \ \&\& \ q$	$p \ \ q$	p	$! \ p$
true	true	true	true	true	false
true	false	false	true	false	true
false	true	false	true		
false	false	false	false		

Por ejemplo, si p es false, y q es true, $p \ \&\& \ q$ será false y $p \ || \ q$ será true.

Recordad la siguiente regla nemotécnica:

▷ false `&&` **expresión** siempre es false.

▷ true `||` **expresión** siempre es true.

Tabla de Precedencia:

!

`&&`

`||`

Ejercicio. Declare dos variables de tipo `bool`, `es_joven` y `es_varon`. Asígneles cualquier valor. Declare otra variable `es_varon_viejo` y asígnele el valor correcto usando las variables anteriores y los operadores lógicos.

I.5.7.3. Operadores Relacionales

Son los operadores habituales de comparación de expresiones numéricas.

Pueden aplicarse a operandos tanto enteros, reales, como de caracteres y tienen el mismo sentido que en Matemáticas. El resultado es de tipo `bool`.

`==` (igual), `!=` (distinto), `<`, `>`, `<=` (menor o igual) y `>=` (mayor o igual)

Algunos ejemplos:

- ▷ La expresión `(4 < 5)` devuelve valor `true`
- ▷ La expresión `(4 > 5)` devuelve el valor `false`
- ▷ La relación de orden entre caracteres se establece según la tabla ASCII (la página de códigos en general)

La expresión `('a' > 'b')` devuelve el valor `false`.

`!=` es el operador relacional distinto.

`!` es la negación lógica.

`==` es el operador relacional de igualdad.

`=` es la operación de asignación.

Tanto en `==` como en `!=` se usan 2 signos para un único operador

```
// Ejemplo de operadores relacionales

int main(){
    int entero1, entero2;
    double real1, real2;
    bool menor, iguales;

    entero1 = 3;
    entero2 = 5;

    menor = entero1 < entero2;           // true
    menor = entero2 < entero1;           // false
    menor = (entero1 < entero2) && !(entero2 < 7); // false
    menor = (entero1 < entero2) || !(entero2 < 7); // true
    iguales = entero1 == entero2;        // false

    real1 = 3.8;
    real2 = 8.1;

    menor = real1 > real2;                // false
    menor = !menor;                       // true
}
```

Veremos su uso en la *sentencia condicional*:

```
if (4 < 5)
    cout << "4 es menor que 5";

if (!(4 > 5))
    cout << "4 es menor o igual que 5";
```

Tabla de Precedencia:

()
!
< <= > >=
== !=
&&
||

A es menor o igual que B y B no es mayor que C

```
int A = 40, B = 34, C = 50;  
bool condicion;
```

```
condicion = A <= B && !B > C;          // Incorrecto  
condicion = (A <= B) && !(B > C);      // Correcto  
condicion = (A <= B) && !(B > C);      // Correcto
```

```
condicion = A <= B && B <= C;    // Correcto. Expresión simplificada
```



Consejo: *Simplifique las expresiones lógicas, para así aumentar su legibilidad.*

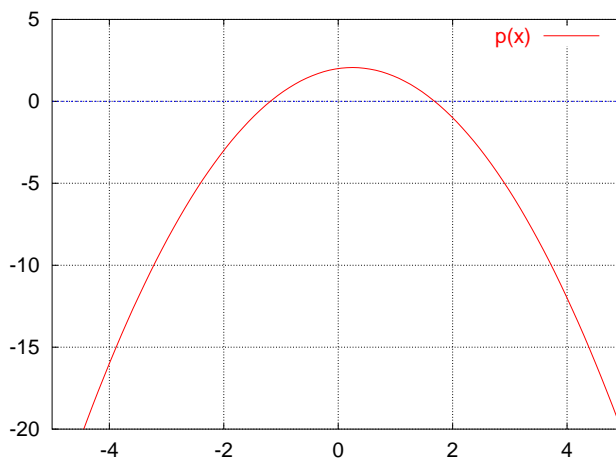


Ejercicio. Escriba una expresión lógica que devuelva `true` si un número entero `edad` está en el intervalo `[0,100]`

I.6. El principio de una única vez

Ejemplo. Calcule las raíces de una ecuación de 2º grado.

$$p(x) = ax^2 + bx + c = 0$$



Algoritmo: Raíces de una parábola

- ▷ **Entradas:** Los parámetros de la ecuación a, b, c .
Salidas: Las raíces de la parábola r_1, r_2
- ▷ **Descripción:**
Calcular r_1, r_2 en la forma siguiente:

$$r_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad r_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$


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

int main(){
    double a, b, c;          // Parámetros de la ecuación
    double raiz1, raiz2;     // Raíces obtenidas

    cout << "\nIntroduce coeficiente de 2º grado: ";
    cin >> a;
    cout << "\nIntroduce coeficiente de 1er grado: ";
    cin >> b;
    cout << "\nIntroduce coeficiente independiente: ";
    cin >> c;

    // Se evalúa dos veces la misma expresión:
    raiz1 = (-b + sqrt(b*b + 4*a*c) ) / (2*a);
    raiz2 = (-b - sqrt(b*b + 4*a*c) ) / (2*a);

    cout << "\nLas raíces son: " << raiz1 << " y " << raiz2;
}
```



En el código anterior se evalúa dos veces la expresión $\text{sqrt}(b*b + 4*a*c)$. Esto es nefasto ya que:

- ▷ **El compilador pierde tiempo al evaluar dos veces una misma expresión. El resultado es el mismo ya que los datos involucrados no han cambiado.**
- ▷ **Mucho más importante: Cualquier cambio que hagamos en el futuro nos obligará a modificar el código en dos sitios distintos. De hecho, había un error en la expresión y deberíamos haber puesto: $b*b - 4*a*c$, por lo que tendremos que modificar dos líneas distintas de nuestro programa.**

Para no repetir código usamos una variable para almacenar el valor de la expresión que se repite:

```
int main(){
    double a, b, c;           // Parámetros de la ecuación
    double raiz1, raiz2;      // Raíces obtenidas
    double radical, denominador;

    cout << "\nIntroduce coeficiente de 2º grado: ";
    cin >> a;
    cout << "\nIntroduce coeficiente de 1er grado: ";
    cin >> b;
    cout << "\nIntroduce coeficiente independiente: ";
    cin >> c;

    // Cada expresión sólo se evalúa una vez:

    denominador = 2*a;
    radical = sqrt(b*b - 4*a*c);

    raiz1 = (-b + radical) / denominador;
    raiz2 = (-b - radical) / denominador;
    cout << "\nLas raíces son: " << raiz1 << " y " << raiz2;
}
```



http://decsai.ugr.es/jccubero/FP/I_ecuacion_segundo_grado.cpp

Nota:

Observe que, realmente, también se repite la expresión $-b$. Debido a la sencillez de la expresión, se ha optado por mantenerla duplicada.

Principio de Programación:

Una única vez (Once and only once)

Cada descripción de comportamiento debe aparecer una única vez en nuestro programa.



O dicho de una manera informal:

El código no debe estar repetido en distintos sitios del programa



La violación de este principio hace que los programas sean difíciles de actualizar ya que cualquier cambio ha de realizarse en todos los sitios en los que está repetido el código. Esto aumenta las posibilidades de cometer un error ya que podría omitirse alguno de estos cambios.

En el tema III (Funciones y Clases) veremos herramientas que los lenguajes de programación proporcionan para poder cumplir este principio. Por ahora, nos limitaremos a seguir el siguiente consejo:

Si el resultado de una expresión no cambia en dos sitios distintos del programa, usaremos una variable para almacenar el resultado de la expresión y utilizaremos su valor tantas veces como queramos.

Bibliografía recomendada para este tema:

- ▷ **A un nivel menor del presentado en las transparencias:**
 - **Primer capítulo de Garrido.**
 - **Primer capítulo de Deitel & Deitel**
- ▷ **A un nivel similar al presentado en las transparencias:**
 - **Capítulo 1 y apartados 2.1, 2.2 y 2.3 de Savitch**
- ▷ **A un nivel con más detalles:**
 - **Los seis primeros capítulos de Breedlove.**
 - **Los tres primeros capítulos de Gaddis.**
 - **Los tres primeros capítulos de Stephen Prata.**
 - **Los dos primeros capítulos de Lafore.**

Resúmenes:

Si el valor de una constante se obtiene a partir de una expresión, no evalúe manualmente dicha expresión: incluya la expresión completa en la definición de la constante.

Evite la evaluación de expresiones en una instrucción de salida de datos. Éstas deben limitarse a imprimir mensajes y el contenido de las variables.

En una sentencia de asignación

`variable = <expresión>`

primero se evalúa la expresión que aparece a la derecha y luego se realiza la asignación.

A lo largo de este tema se verán operadores y funciones aplicables a los distintos tipos de datos. No es necesario aprenderse el nombre de todos ellos pero sí saber cómo se usan.

A un dato numérico se le puede asignar una expresión de un tipo distinto. Si el resultado cabe, no hay problema. En otro caso, se produce un desbordamiento aritmético y se asigna un valor indeterminado. Es un error lógico, pero no se produce un error de ejecución.

Si asignamos una expresión real a un entero, se trunca la parte decimal.

Si en una expresión todos los datos son del mismo tipo, la expresión devuelve un dato de ese tipo.

Durante la evaluación de una expresión numérica en la que intervienen datos de distinto tipo, el compilador realizará un casting para transformar los datos de tipos pequeños al mayor de los tipos involucrados.

Esta transformación es temporal (sólo se aplica mientras se evalúa la expresión).

Pero cuidado: si en una expresión todos los datos son del mismo tipo, el compilador no realiza ninguna transformación, de forma que la expresión resultante es del mismo tipo que la de los datos involucrados. Por tanto, cabe la posibilidad que se produzca un desbordamiento durante la evaluación de la expresión.

El tipo de dato `char` es un entero usualmente pequeño que permite albergar al menos 256 valores distintos (así se indica en C++ 14) y está pensado para realizar operaciones de E/S con caracteres.

Si el resultado de una expresión no cambia en dos sitios distintos del programa, usaremos una variable para almacenar el resultado de la expresión y utilizaremos su valor tantas veces como queramos.

Consejo: Para facilitar la lectura de las fórmulas matemáticas, evite el uso de paréntesis cuando esté claro cuál es la precedencia de cada operador.



Consejo: Simplifique las expresiones lógicas, para así aumentar su legibilidad.



Evite siempre el uso de *números mágicos* (*magic numbers*), es decir, literales numéricos cuyo significado en el código no queda claro.

Fomentaremos el uso de datos constantes en vez de literales para representar toda aquella información que sea constante durante la ejecución del programa.

IMPORTANT

Todas las operaciones realizadas con datos de tipo real pueden devolver valores que sólo sean aproximados

IMPORTANT

Cada entidad que vaya a representar en un programa debe estar definida en un único sitio. Dicho de otra forma, no use varias variables para representar un mismo concepto.

IMPORTANT

Para hacer más legible el código fuente, usaremos separadores como el espacio en blanco, el tabulador y el retorno de carro



Como programadores profesionales, debemos seguir las normas de codificación de código establecidas en la empresa.



El código no debe estar repetido en distintos sitios del programa



Principio de Programación:

Una única vez (Once and only once)

Cada descripción de comportamiento debe aparecer una única vez en nuestro programa.



Tema II

Estructuras de control

Objetivos:

- ▷ Introducir las estructuras condicionales que nos permitirán realizar saltos hacia adelante durante la ejecución del código.
- ▷ Introducir las estructuras repetitivas que nos permitirán realizar saltos hacia atrás durante la ejecución del código.
- ▷ Introducir pautas y buenos hábitos de programación en la construcción de las estructuras condicionales y repetitivas.

II.1. Estructura condicional

II.1.1. Flujo de control

El *flujo de control (control flow)* es la especificación del orden de ejecución de las sentencias de un programa.

Una forma de especificarlo es numerando las líneas de un programa. Por ejemplo, numeremos las sentencias del ejemplo de la página 111 (excepto las de declaración de los datos):

```
int main(){
    double a, b, c;           // Parámetros de la ecuación
    double raiz1, raiz2;      // Raíces obtenidas
    double radical, denominador;

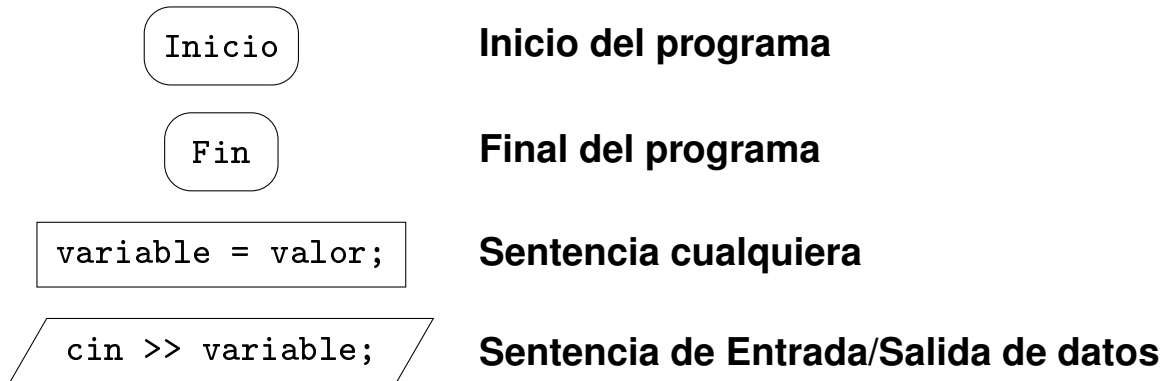
1  cout << "\nIntroduce coeficiente de 2º grado: ";
2  cin >> a;
3  cout << "\nIntroduce coeficiente de 1er grado: ";
4  cin >> b;
5  cout << "\nIntroduce coeficiente independiente: ";
6  cin >> c;

7  denominador = 2*a;
8  radical = sqrt(b*b - 4*a*c);
9  raiz1 = (-b + radical) / denominador;
10 raiz2 = (-b - radical) / denominador;

11 cout << "\nLas raíces son" << raiz1 << " y " << raiz2;
}
```

Flujo de control: (1,2,3,4,5,6,7,8,9,10,11)

Otra forma de especificar el orden de ejecución de las sentencias es usando un *diagrama de flujo (flowchart)* . Los símbolos básicos de éste son:



Hasta ahora, el orden de ejecución de las sentencias es secuencial, es decir, éstas se van ejecutando sucesivamente, siguiendo su orden de aparición. Esta forma predeterminada de flujo de control diremos que constituye la **estructura secuencial (sequential control flow structure)**.

En los siguientes apartados vamos a ver cómo realizar saltos. Éstos podrán ser:

▷ **Hacia delante.**

Implicará que un conjunto de sentencias no se ejecutarán.

Vienen definidos a través de una estructura condicional.

▷ **Hacia atrás.**

Implicará que volverá a ejecutarse un conjunto de sentencias.

Vienen definidos a través de una estructura repetitiva.

¿Cómo se realiza un salto hacia delante? Vendrá determinado por el cumplimiento de una **condición (condition)** especificada a través de una expresión lógica.

Una **estructura condicional (conditional structure)** es una estructura que permite la ejecución de una (o más) sentencia(s) dependiendo de la evaluación de una condición.

Existen tres tipos: *Simple*, *Doble* y *Múltiple*

II.1.2. Estructura condicional simple

II.1.2.1. Formato

```
if (<condición>
    <bloque if>
```

<condición> es una expresión lógica

<bloque if> es el bloque que se ejecutará si la expresión lógica se evalúa a `true`. Si hay varias sentencias dentro, es necesario encerrarlas entre llaves. Si sólo hay una sentencia, pueden omitirse las llaves.

Los paréntesis que encierran la condición son obligatorios.

Todo el bloque que empieza por `if` y termina con la última sentencia dentro del condicional (incluida la llave cerrada, en su caso), forma una única sentencia, denominada *sentencia condicional (conditional statement)* .

Ejemplo. Continuando el ejemplo de la página 119



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

int main(){
    double a, b, c;           // Parámetros de la ecuación
    double raiz1, raiz2;      // Raíces obtenidas
    double radical, denominador;

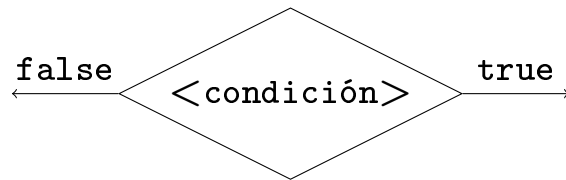
    cout << "\nIntroduce coeficiente de 2º grado: ";
    cin >> a;
    cout << "\nIntroduce coeficiente de 1er grado: ";
    cin >> b;
    cout << "\nIntroduce coeficiente independiente: ";
    cin >> c;

    if (a != 0) {
        denominador = 2*a;
        radical = sqrt(b*b - 4*a*c);
        raiz1 = (-b + radical) / denominador;
        raiz2 = (-b - radical) / denominador;

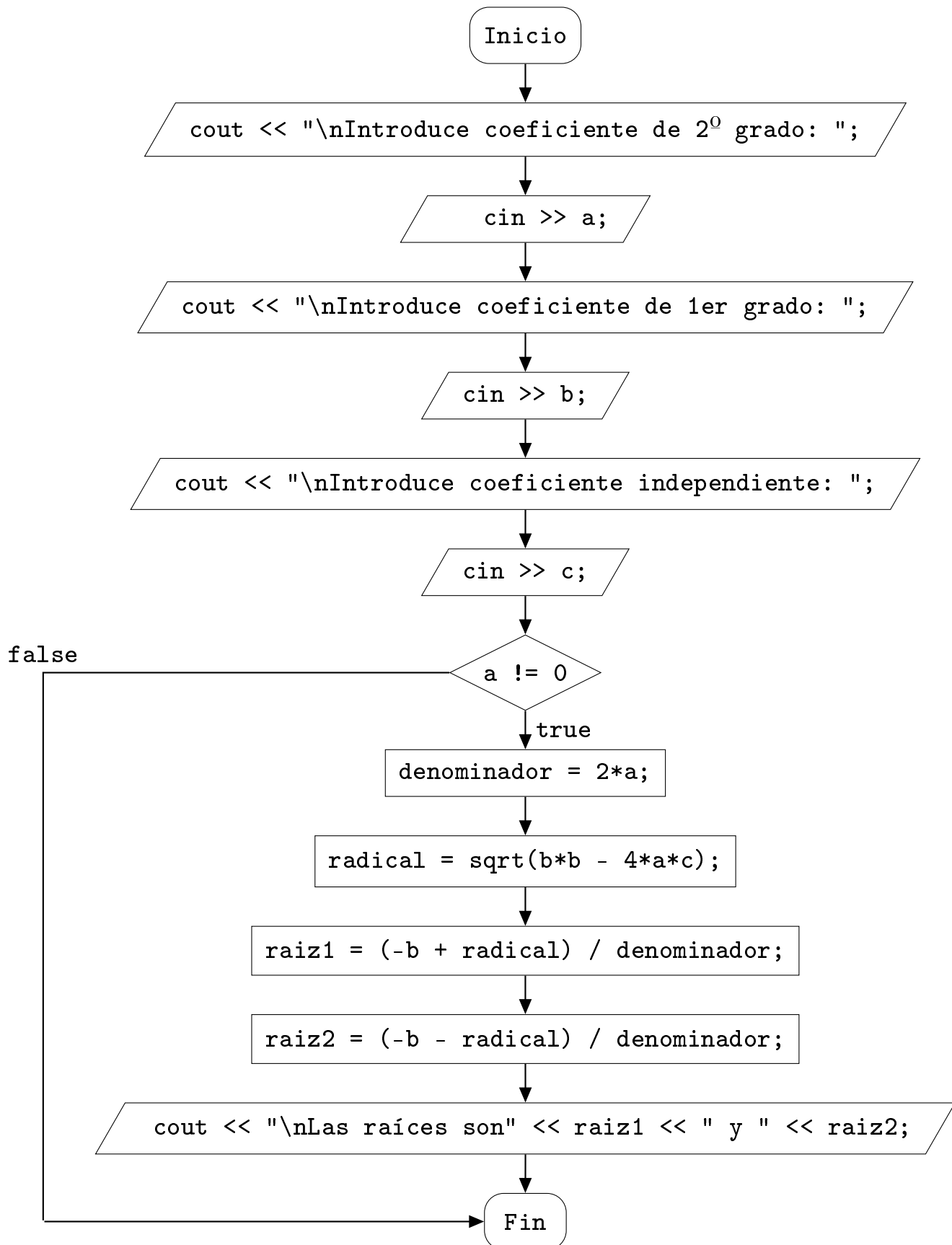
        cout << "\nLas raíces son" << raiz1 << " y " << raiz2;
    }
}
```

II.1.2.2. Diagrama de flujo

Para representar una estructura condicional en un diagrama de flujo, se utiliza un rombo:



El ejemplo de la ecuación de segundo grado quedaría:



Si hay varias sentencias, es necesario encerrarlas entre llaves. Dicho de otra forma: si no ponemos llaves, el compilador entiende que la única sentencia del bloque `if` es la que hay justo debajo.

```
if (a != 0)
    denominador = 2*a;
    radical = sqrt(b*b - 4*a*c);

    raiz1 = (-b + radical) / denominador;
    raiz2 = (-b - radical) / denominador;

    cout << "\nLas raíces son" << raiz1 << " y " << raiz2;
```



Para el compilador es como si fuese:

```
if (a != 0){
    denominador = 2*a;
}

radical = sqrt(b*b - 4*a*c);

raiz1 = (-b + radical) / denominador;
raiz2 = (-b - radical) / denominador;

cout << "\nLas raíces son" << raiz1 << " y " << raiz2;
```

Ejemplo. Lea un número e imprima "Es par" en el caso de que sea par.

```
int entero;
cin >> entero;

if (entero % 2 == 0){
    cout << "\nEs par";
}

cout << "\nFin del programa";
```

o si se prefiere:

```
if (entero % 2 == 0)
    cout << "\nEs par";

cout << "\nFin del programa";
```

Si el número es impar, únicamente se mostrará el mensaje:

```
Fin del programa
```

Ejemplo. Lea una variable `salario` y si éste es menor de 1300 euros, imprima en pantalla el mensaje "Tiene un salario bajo"

```
double salario;  
  
if (salario < 1300)  
    cout << "\nTiene un salario bajo";  
  
cout << "\nFin del programa";
```

Con un salario de 900 euros, por ejemplo, en pantalla veremos:

```
Tiene un salario bajo  
Fin del programa
```

Con un salario de 2000 euros, por ejemplo, en pantalla veremos:

```
Fin del programa
```

II.1.2.3. Variables no asignadas en los condicionales

Supongamos que una variable no tiene un valor asignado antes de entrar a un condicional. ¿Qué ocurre si dentro del bloque `if` le asignamos un valor pero no lo hacemos en el bloque `else`? Si la condición es `false`, al salir del condicional, la variable seguiría teniendo un valor indeterminado.

Ejemplo. Lea dos variables enteras `a` y `b` y asigne a una variable `max` el máximo de ambas.

```
int a, b, max;
```

```
cin >> a;
```

```
cin >> b;
```

```
if (a > b)
    max = a;
```



¿Qué ocurre si la condición es falsa? La variable `max` se queda sin ningún valor asignado.

Una posible solución: La inicializamos a un valor por defecto antes de entrar al condicional. Veremos otra solución cuando introduzcamos el condicional doble.

```
int a, b, max;
```

```
cin >> a;
```

```
cin >> b;
```

```
max = b;
```

```
if (a > b)
    max = a;
```



Ejemplo. Lea el valor de la edad y salario de una persona. Súbale el salario un 4% si tiene más de 45 años.

```
int edad;  
double salario_base, salario_final;  
  
cout << "\nIntroduzca edad y salario ";  
cin  >> edad;  
cin  >> salario_base;  
  
salario_final = salario_base;  
  
if (edad >= 45)  
    salario_final = salario_final * 1.04;
```



En el caso de que la edad sea menor de 45, el salario final no se modificará y se quedará con el valor previamente asignado (`salario_base`)

Debemos prestar especial atención a los condicionales en los que se asigna un primer valor a alguna variable. Intentaremos garantizar que dicha variable salga siempre del condicional (independientemente de si la condición era verdadera o falsa) con un valor establecido.

II.1.2.4. Cuestión de estilo

El compilador se salta todos los separadores (espacios, tabulaciones, etc) entre las sentencias delimitadas por ;

Para favorecer la lectura del código y enfatizar el bloque de sentencias incluidas en la estructura condicional, usaremos el siguiente estilo de codificación:

```
cin >> c;

                                // <- Línea en blanco antes del condicional
if (a != 0){
    denominador = 2*a;        // Líneas tabuladas con 3 espacios
    radical = sqrt(b*b - 4*a*c);
    raiz1 = (-b + radical) / denominador;
    raiz2 = (-b - radical) / denominador;

    cout << "\nLas raíces son" << raiz1 << " y " << raiz2;
}

                                // <- Línea en blanco después del condicional
```

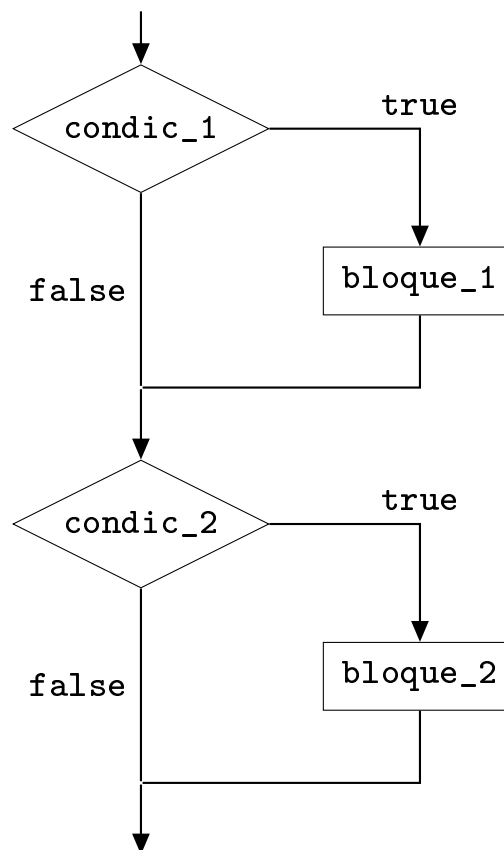
Destaque visualmente el bloque de instrucciones de una estructura condicional.

IMPORTANT

II.1.2.5. Estructuras condicionales consecutivas

¿Qué ocurre si en el código hay dos estructuras condicionales simples consecutivas?

```
if (condic_1)
    bloque_1
if (condic_2)
    bloque_2
```



La ejecución de `bloque_1` sólo depende de `condic_1` (idem con el otro bloque) Por tanto, podrán ejecutarse ambos bloques, ninguno o cualquiera de ellos.

Usaremos estructuras condicionales consecutivas cuando los criterios de cada una de ellas sean independientes del resto.

Ejemplo. Compruebe si una persona es mayor de edad y si tiene más de 190 cm de altura. Si bien es cierto que es difícil que un menor de edad mida más de 190 cm, no es una situación imposible. Así pues, podemos considerar que las condiciones son independientes y por tanto usamos dos condicionales consecutivos.

```
int edad, altura;
cin >> edad;
cin >> altura;

if (edad >= 18)
    cout << "\nEs mayor de edad";

if (altura >= 190)
    cout << "\nEs alto/a";
```



Dependiendo de los valores de `edad` y `altura` se pueden imprimir ambos mensajes, uno sólo de ellos o ninguno.

Ejercicio. Retome el ejemplo de la subida salarial de la página 130:

```
salario_final = salario_base;

if (edad >= 45)
    salario_final = salario_final * 1.04;
```

Si la persona tiene más de dos hijos, súbale un 2% adicional. Esta subida es independiente de la subidad salarial del 4% por la edad. En el caso de que ambos criterios sean aplicables, la subida del 2% se realizará sobre el resultado de haber incrementado previamente el sueldo el 4%.

II.1.2.6. Condiciones compuestas

En muchos casos tendremos que utilizar condiciones compuestas:

Ejemplo. Lea el valor de la edad y salario de una persona. Súbale el salario un 4% si tiene más de 45 años y además gana menos de 1300 euros.

```
int edad;
double salario_base, salario_final;

cout << "\nIntroduzca edad y salario ";
cin  >> edad;
cin  >> salario_base;

salario_final = salario_base;

if (edad >= 45 && salario_base < 1300)
    salario_final = salario_final * 1.04;
```



Ejemplo. Cambie el ejercicio anterior para subir el salario si tiene más de 45 años o si gana menos de 1300 euros.

```
.....
salario_final = salario_base;

if (edad >= 45 || salario_base < 1300)
    salario_final = salario_final * 1.04;
```



Si cualquiera de las dos desigualdades es verdadera, se incrementará el salario.

Ejercicio. Compruebe si una persona es menor de edad o mayor de 65 años.

Ejemplo. Compruebe si un número real está dentro de un intervalo cerrado [inferior, superior].

```
double inferior, superior, dato;
```

```
cout << "\nIntroduzca los extremos del intervalo: ";
```

```
cin >> inferior;
```

```
cin >> superior;
```

```
cout << "\nIntroduzca un real arbitrario: ";
```

```
cin >> dato;
```

```
if (dato >= inferior && dato <= superior)
```

```
    cout << "\nEl valor " << dato << " está dentro del intervalo";
```



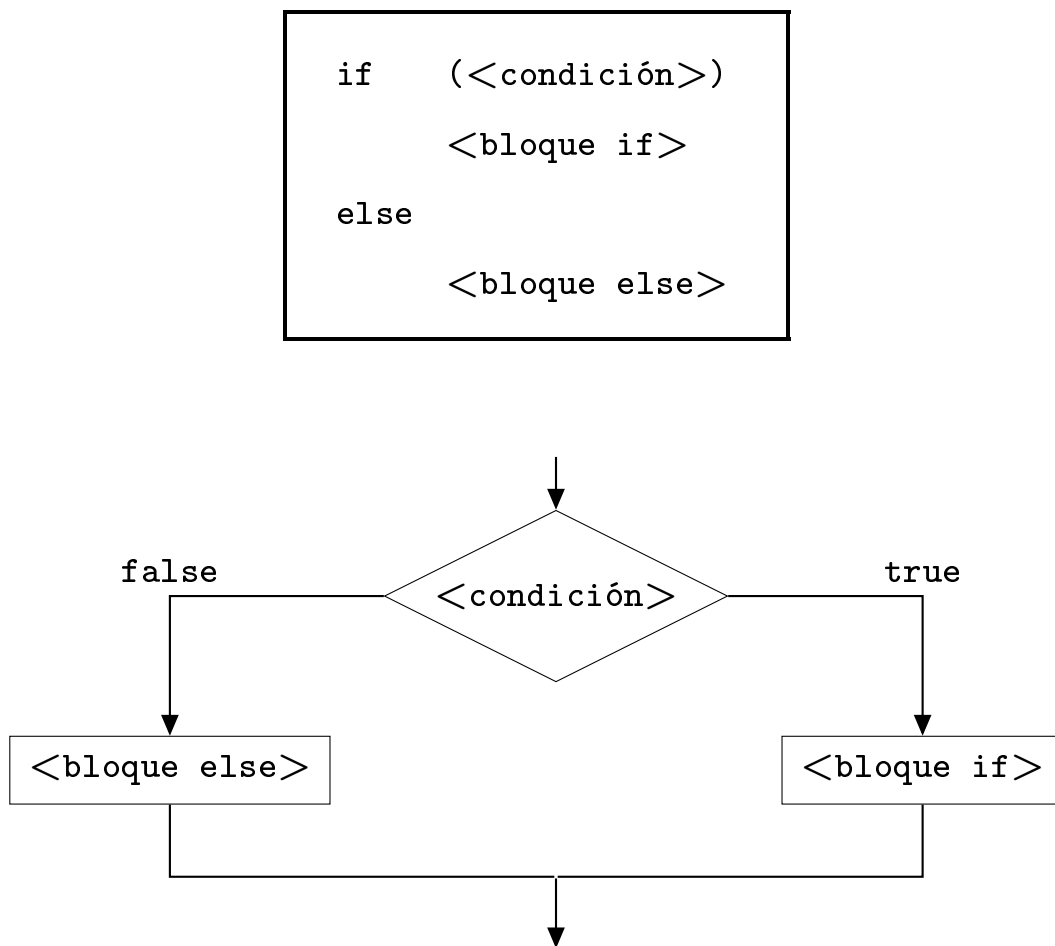
136

Ejercicio. Retome el ejemplo anterior del intervalo y asígnele el valor correcto a una variable de tipo `bool pertenece_al_intervalo`:

II.1.3. Estructura condicional doble

II.1.3.1. Formato

En numerosas ocasiones queremos realizar una acción en el caso de que una condición sea verdadera y otra acción distinta en cualquier otro caso. Para ello, usaremos la **estructura condicional doble** (*else conditional structure*) :



Todo el bloque que empieza por `if` y termina con la última sentencia incluida en el `else` (incluida la llave cerrada, en su caso), forma una única sentencia, denominada **sentencia condicional doble** (*else conditional statement*) .

Ejemplo. ¿Qué pasa si $a = 0$ en la ecuación de segundo grado? El algoritmo debe devolver $-c/b$.



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

int main(){
    double a, b, c;          // Parámetros de la ecuación
    double raiz1, raiz2;     // Raíces obtenidas
    double radical, denominador;

    cout << "\nIntroduce coeficiente de 2º grado: ";
    cin >> a;
    cout << "\nIntroduce coeficiente de 1er grado: ";
    cin >> b;
    cout << "\nIntroduce coeficiente independiente: ";
    cin >> c;

    if (a != 0) {
        denominador = 2*a;
        radical = sqrt(b*b - 4*a*c);

        raiz1 = (-b + radical) / denominador;
        raiz2 = (-b - radical) / denominador;

        cout << "\nLas raíces son" << raiz1 << " y " << raiz2;
    }
    else{
        raiz1 = -c/b;
        cout << "\nLa única raíz es " << raiz1;
    }
}
```

Ejemplo. Compruebe si un número es par (continuación)

```
cin >> entero;

if (entero % 2 == 0)
    cout << "\nEs par";
else
    cout << "\nEs impar";

cout << "\nFin del programa";
```

Siempre imprimirá dos mensajes. Uno relativo a la condición de ser par o impar y luego el del fin del programa.

Ejemplo. Compruebe si el salario es bajo (continuación)

```
double salario;

if (salario < 1300)
    cout << "\nTiene un salario bajo";
else
    cout << "\nNo tiene un salario bajo";

cout << "\nFin del programa";
```

Ejercicio. Máximo de dos valores. Lo resolvimos en la página 129 de la siguiente forma:

```
max = b;  
  
if (a > b)  
    max = a;
```

Resuélvalo ahora utilizando un condicional doble (ambas formas serían correctas)

Ejemplo. Retomemos el ejemplo de la página 135:

```
salario_final = salario_base;  
  
if (edad >= 45 || salario_base < 1300)  
    salario_final = salario_final * 1.04;
```

¿Qué ocurre si la condición es falsa? ¿Necesitamos el siguiente condicional doble?:

```
if (edad >= 45 || salario_base < 1300)  
    salario_final = salario_final * 1.04;  
else  
    salario_final = salario_final;
```



Obviamente no. En este caso nos quedamos con la primera versión, usando un condicional simple que se encarga de *actualizar*, en su caso, la variable `salario_final`.

Ejemplo. Valor dentro de un intervalo (continuación de lo visto en la página 136)

```
pertenece_al_intervalo = false;

if (dato >= inferior && dato <= superior)
    pertenece_al_intervalo = true;
```

En vez de inicializar el valor de `pertenece_al_intervalo` antes de entrar al condicional, le podemos asignar un valor en el `else`

```
if (dato >= inferior && dato <= superior)
    pertenece_al_intervalo = true;
else
    pertenece_al_intervalo = false;
```

Observe que podemos resumir la anterior estructura condicional en una única sentencia, que posiblemente sea más clara:

```
pertenece_al_intervalo = (dato >= inferior && dato <= superior);
```



Consejo: *En aquellos casos en los que debe asignarle un valor a una variable `bool` dependiendo del resultado de la evaluación de una expresión lógica, utilice directamente la asignación de dicha expresión en vez de un condicional doble.*



Ejemplo. Complete el ejercicio anterior e imprima el mensaje adecuado después de la asignación de la variable `pertenece_al_intervalo`.

```
pertenece_al_intervalo = dato >= inferior && dato <= superior;

if (pertenece_al_intervalo)
    cout << "El valor está dentro del intervalo";
else
    cout << "El valor está fuera del intervalo";
```

Observe que no es necesario poner

```
if (pertenece_al_intervalo == true)
```

Basta con poner:

```
if (pertenece_al_intervalo)
```

Consejo: *En los condicionales que simplemente comprueban si una variable lógica contiene `true`, utilice el formato `if (variable_logica)`. Si se quiere consultar si la variable contiene `false` basta poner `if (!variable_logica)`*



II.1.3.2. Condiciones mutuamente excluyentes

El ejemplo del número par (página 139) podría haberse resuelto usando dos condicionales simples consecutivos, en vez de un condicional doble:

```
if (entero % 2 == 0)
    cout << "\nEs par";

if (entero % 2 != 0)
    cout << "\nEs impar";

cout << "\nFin del programa";
```



Como las dos condiciones `entero % 2 == 0` y `entero % 2 != 0` no pueden ser verdad simultáneamente, garantizamos que no se muestren los dos mensajes consecutivamente. Esta solución funciona pero, aunque no lo parezca, repite código:

`entero % 2 == 0` es equivalente a `!(entero % 2 != 0)`

Por lo tanto, el anterior código es equivalente al siguiente:

```
if (entero % 2 == 0)
    cout << "\nEs par";

if (!(entero % 2 == 0))
    cout << "\nEs impar";
```



Ahora se observa mejor que estamos repitiendo código, violando por tanto el principio de una única vez. Por esta razón, en estos casos, debemos utilizar la estructura condicional doble en vez de dos condicionales simples consecutivos.

Ejemplo. La solución al ejemplo de la ecuación de segundo grado utilizando dos condicionales simples consecutivos sería:

```
if (a != 0){  
    denominador = 2*a;  
    radical = sqrt(b*b - 4*a*c);  
  
    raiz1 = (-b + radical) / denominador;  
    raiz2 = (-b - radical) / denominador;  
  
    cout << "\nLas raíces son" << raiz1 << " y " << raiz2;  
}  
  
if (a == 0)  
    cout << "\nTiene una única raíz" << -c/b;
```



Al igual que antes, estaríamos repitiendo código por lo que debemos usar la solución con el condicional doble.

En Lógica y Estadística se dice que un conjunto de dos sucesos es **mutuamente excluyente** (*mutually exclusive*) si se verifica que cuando uno cualquiera de ellos es verdadero, el otro es falso. Por ejemplo, obtener cara o cruz al lanzar una moneda al aire, o ser mayor o menor de edad.

Por extensión, diremos que las condiciones que definen los sucesos son mutuamente excluyentes.

Ejemplo.

- ▷ Las condiciones `entero % 2 == 0` y `entero % 2 != 0` son mutuamente excluyentes: si la expresión `entero % 2 == 0` es `true`, la expresión `entero % 2 != 0` siempre es `false` y viceversa.
- ▷ Las condiciones `(a != 0)` y `(a == 0)` son mutuamente excluyentes.
- ▷ Las condiciones `(edad >= 18)` y `(edad < 18)` son mutuamente excluyentes.

```
if (entero % 2 == 0)
    .....
if (entero % 2 != 0)
    .....
```

```
if (edad >= 18)
    .....
if (edad < 18)
    .....
```

```
if (a != 0)
    .....
if (a == 0)
    .....
```



```
if (entero % 2 == 0)
    .....
else
    .....
```

```
if (edad >= 18)
    .....
else
    .....
```

```
if (a != 0)
    .....
else
    .....
```



Cuando trabajamos con expresiones compuestas, se hace aún más patente la necesidad de usar un condicional doble. Veámoslo con el siguiente ejemplo.

Ejemplo. Retome el ejemplo de la subida salarial de la página 134. Si no se cumple la condición de la subida del 4%, el salario únicamente se incrementará en un 1%.

```
edad >= 45 && salario_base < 1300 -> +4%  
Otro caso                          -> +1%
```

¿Cuándo se produce la subida del 1%? Cuando la expresión `edad >= 45 && salario_base < 1300` es false, es decir, cuando cualquiera de ellos es false. Nos quedaría:

```
edad >= 45 && salario_base < 1300 -> +4%  
edad < 45 || salario_base >= 1300 -> +1%
```

¿Sería correcto entonces poner lo siguiente?

```
salario_final = salario_base;  
  
if (edad >= 45 && salario_base < 1300)  
    salario_final = salario_final * 1.04;  
if (edad < 45 || salario_base >= 1300) // Código repetido  
    salario_final = salario_final * 1.01;
```



El resultado, de cara al usuario, sería el mismo, pero es un código nefasto, ya que se viola el principio de una única vez. Por lo tanto, usamos mucho mejor una estructura condicional doble:

```
salario_final = salario_base;  
  
if (edad >= 45 && salario_base < 1300)  
    salario_final = salario_final * 1.04;  
else  
    salario_final = salario_final * 1.01;
```



En resumen:

La estructura condicional doble nos permite trabajar con dos condiciones mutuamente excluyentes, comprobando únicamente una de ellas en la parte del `if`. Esto nos permite cumplir el principio de una única vez.

<code>if (cond)</code>		<code>if (cond)</code>
<code>...</code>		<code>...</code>
<code>if (!cond)</code>	→	<code>else</code>
<code>...</code>		<code>...</code>

II.1.3.3. Álgebra de Boole

Debemos intentar simplificar las expresiones lógicas, para que sean fáciles de entender. Usaremos las propiedades del *álgebra de Boole (Boolean algebra)* :

<code>!(A B)</code>	equivale a	<code>!A && !B</code>
<code>!(A && B)</code>	equivale a	<code>!A !B</code>
<code>A && (B C)</code>	equivale a	<code>(A && B) (A && C)</code>
<code>A (B && C)</code>	equivale a	<code>(A B) && (A C)</code>

Nota. Las dos primeras (relativas a la negación) se conocen como *Leyes de De Morgan (De Morgan's laws)* . Las dos siguientes son la ley distributiva (de la conjunción con respecto a la disyunción y viceversa).

Ampliación:

Consulte:

http://es.wikipedia.org/wiki/Algebra_booleana

<http://serbal.pntic.mec.es/~cmunoz11/boole.pdf>



Ejemplo. Es un contribuyente especial si la edad está entre 16 y 65 y sus ingresos están por debajo de 7000 o por encima de 75000 euros. La expresión lógica sería:

```
(16 <= edad && edad <= 65 && ingresos < 7000)
||
(16 <= edad && edad <= 65 && ingresos > 75000)
```

Para simplificar la expresión, sacamos factor común y aplicamos la ley distributiva:

```
(16 <= edad && edad <= 65)
&&
(ingresos < 7000 || ingresos > 75000)
```

De esta forma, no repetimos la expresión `edad >= 16 && edad <= 65`

En resumen:

Utilice las leyes del Álgebra de Boole para simplificar las expresiones lógicas.

Ejemplo. Un trabajador es un aprendiz si su edad no está por debajo de 16 (estricto) o por encima de 18. La expresión lógica que debemos usar es:

```
!(edad < 16 || 18 <= edad)      // Difícil de entender
```

Aplicando las leyes de Morgan:

```
!(edad < 16 || 18 <= edad)  
    equivale a  
!(edad < 16) && !(18 <= edad)  
    equivale a  
(16 <= edad) && (edad < 18)
```

Realmente no son necesarios los paréntesis de las expresiones ya que el operador && tiene menos prioridad. Nos quedaría finalmente:

```
16 <= edad && edad < 18      // Más fácil de entender
```

Ejemplo. Suba un 4% el salario del trabajador si tiene más de 45 años. Si no los tiene, también se subirá siempre y cuando tenga más de 3 años de experiencia.

```
salario_final = salario_base;

if (edad >= 45 || (edad < 45 && experiencia > 3))
    salario_final = salario_final * 1.04;
```

Observe que `edad >= 45` **equivale a** `!(edad < 45)` **por lo que estamos ante una expresión del tipo**

`A || (!A && B)`

Veamos que esta expresión se puede simplificar a la siguiente:

`A || B`

Aplicamos la ley distributiva:

`A || (!A && B) <-> (A || !A) && (A || B)`
`<->`
`True && (A || B) <-> A || B`

Aplicándolo a nuestro ejemplo, nos quedaría finalmente:

```
salario_final = salario_base;

if (edad >= 45 || experiencia > 3)
    salario_final = salario_final * 1.04;
```

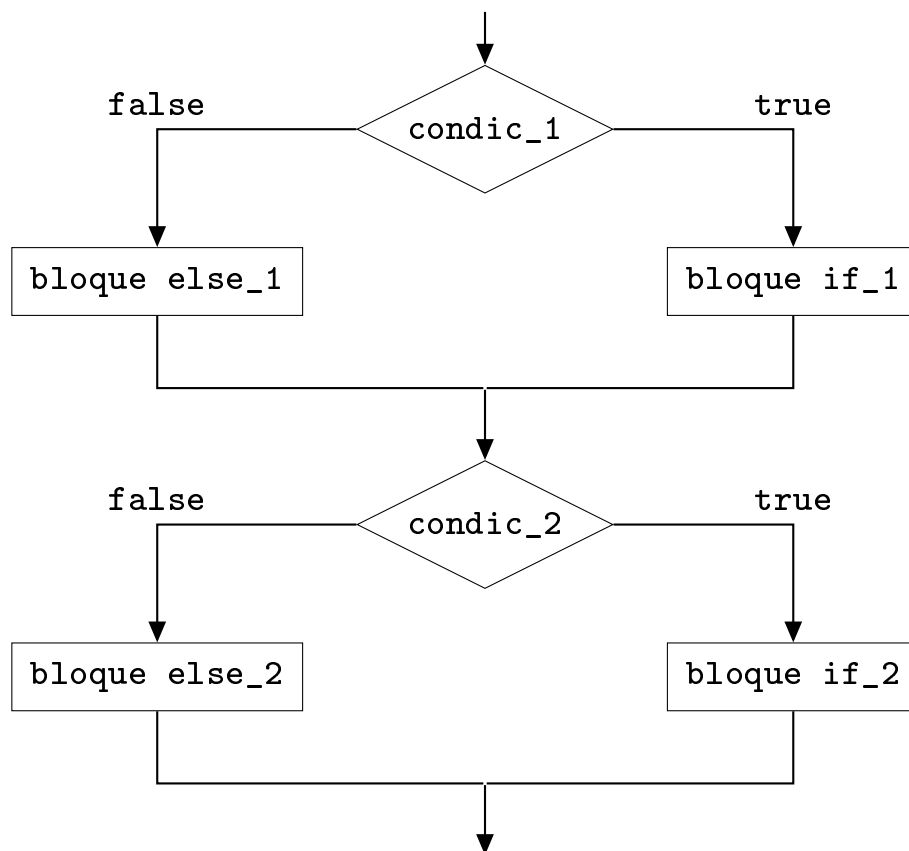
La expresión `A || (!A && B)` ***la sustituiremos por la equivalente a ella:*** `A || B`

II.1.3.4. Estructuras condicionales dobles consecutivas

Un condicional doble garantiza la ejecución de un bloque de instrucciones: o bien el bloque del `if`, o bien el bloque del `else`. Si hay dos condicionales dobles consecutivos, se ejecutarán los bloques de instrucciones que corresponda: o ninguno, o uno de ellos, o los dos.

```
if (condic_1)
    bloque if_1
else
    bloque else_1
```

```
if (condic_2)
    bloque if_2
else
    bloque else_2
```



Veamos un ejemplo de un condicional doble y otro simple consecutivos.

Ejemplo. Retome el ejemplo de la subida salarial de la página 146. Considere otro criterio en la subida salarial: si tiene más de dos hijos, se subirá un 2% adicional sobre el salario incrementado anteriormente.

```
edad >= 45 && salario_base < 1300  ->  +4%
Otro caso                          ->  +1%

numero_hijos > 2                  ->  +2%
```

Nos quedaría:

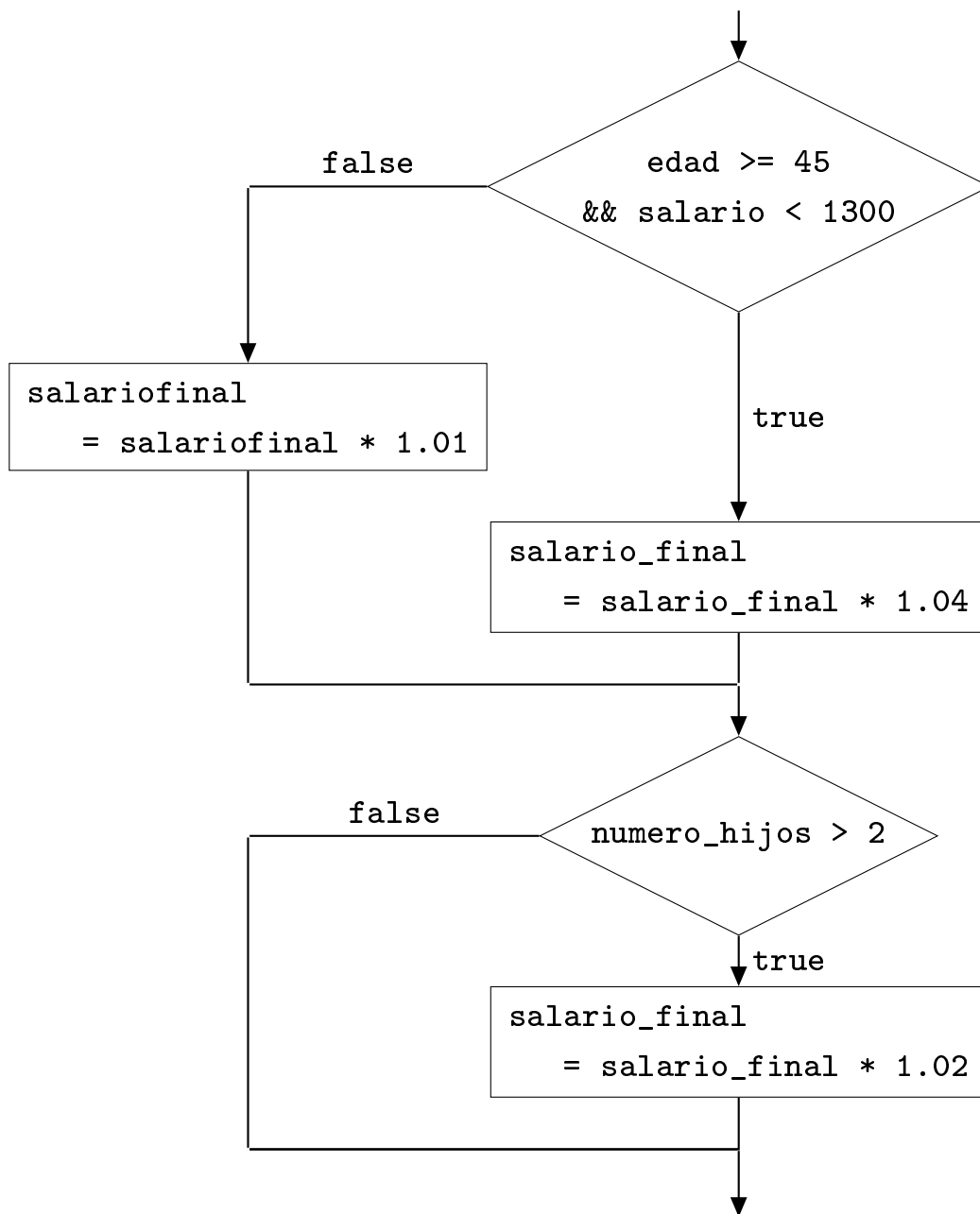
```
.....
salario_final = salario_base;

if (edad >= 45 && salario_base < 1300)
    salario_final = salario_final * 1.04;
else
    salario_final = salario_final * 1.01;

if (numero_hijos > 2)
    salario_final = salario_final * 1.02;
```



Destaquemos que la segunda subida se realiza sobre el salario final ya modificado en el primer condicional. Se deja como ejercicio realizar la segunda subida tomando como referencia el salario base.



Veamos un ejemplo de dos condicionales dobles consecutivos.

Ejemplo. Lea la edad y la altura de una persona y diga si es mayor o menor de edad y si es alta o no. Se pueden dar las cuatro combinaciones posibles: mayor de edad alto, mayor de edad no alto, menor de edad alto, menor de edad no alto.

```
int edad, altura;

cin >> edad;
cin >> altura;

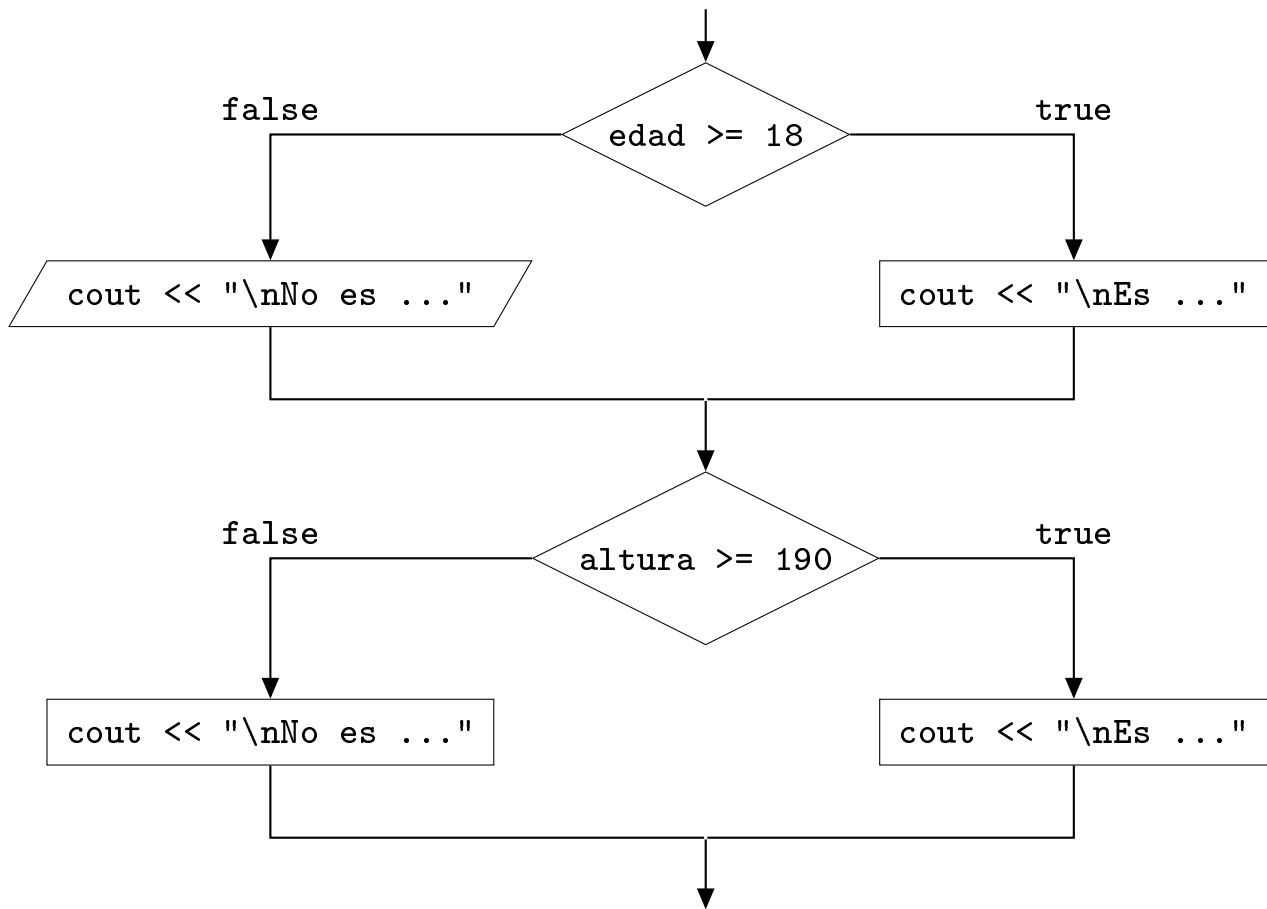
if (edad >= 18)
    cout << "\nEs mayor de edad";
else
    cout << "\nEs menor de edad";

if (altura >= 190)
    cout << "\nEs alto/a";
else
    cout << "\nNo es alto/a";
```



Siempre se imprimirán dos mensajes. Uno relativo a la condición de ser mayor o menor de edad y el otro relativo a la altura.

Una situación típica de uso de estructuras condicionales dobles consecutivas, se presenta cuando tenemos que comprobar distintas condiciones independientes entre sí. Dadas n condiciones que dan lugar a n condicionales dobles consecutivos, se pueden presentar 2^n situaciones posibles.



Tal y como veíamos en la página 143, si hubiésemos duplicado las expresiones de los condicionales habría mucho código repetido y estaríamos violando el principio de una única vez:

```
if (edad >= 18 && altura >= 190)
    cout << "\nEs mayor de edad" << "\nEs alto/a";

if (edad >= 18 && altura < 190)
    cout << "\nEs mayor de edad" << "\nNo es alto/a";

if (edad < 18 && altura >= 190)
    cout << "\nEs menor de edad" << "\nEs alto/a";

if (edad < 18 && altura < 190)
    cout << "\nEs menor de edad" << "\nNo es alto/a";
```

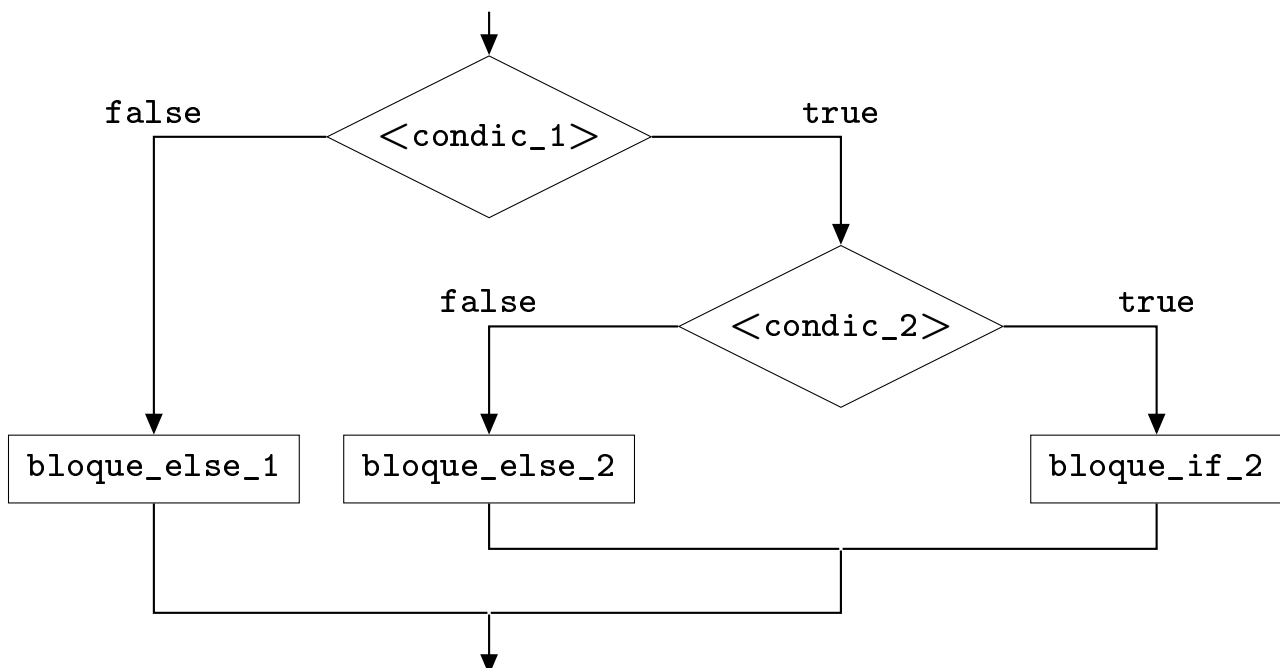


II.1.4. Anidamiento de estructuras condicionales

II.1.4.1. Funcionamiento del anidamiento

Dentro de un bloque `if` o `else`, puede incluirse otra estructura condicional, anidándose tanto como permita el compilador (aunque lo normal será que no tengamos más de tres o cuatro niveles de anidamiento). Por ejemplo, si anidamos un condicional doble dentro de un bloque `if`:

```
if (condic_1){  
    if (condic_2){  
        bloque_if_2  
    }  
    else{  
        bloque_else_2  
    }  
else{  
    bloque_else_1  
}
```



Ejemplo. ¿Cuándo se ejecuta cada instrucción?

```
if (condic_1){  
    inst_1;  
  
    if (condic_2){  
        inst_2;  
    }  
    else{  
        inst_3;  
    }  
  
    inst_4;  
}  
else{  
    inst_5;  
  
    if (condic_3)  
        inst_6;  
}
```

	condic_1	condic_2	condic_3
inst_1	true	da igual	da igual
inst_2	true	true	da igual
inst_3	true	false	da igual
inst_4	true	da igual	da igual
inst_5	false	da igual	da igual
inst_6	false	da igual	true

Recordemos que las estructuras condicionales consecutivas se utilizan cuando los criterios de las condiciones son independientes. Por contra, Usaremos los condicionales anidados cuando los criterios sean dependientes entre sí.

Ejemplo. Cambiamos el ejemplo de la página 155 para implementar el siguiente criterio: Si es mayor de edad, el umbral para decidir si una persona es alta o no es 190 cm. Si es menor de edad, el umbral baja a 175 cm.

Ahora, el criterio por el que determinamos si una persona es alta depende de la edad. Por tanto, en vez de usar dos condicionales consecutivos como hicimos en el ejemplo de la página 155, ahora debemos usar condicionales anidados:

```
int edad, altura;

cin >> edad;
cin >> altura;

if (edad >= 18){
    cout << "\nEs mayor de edad";

    if (altura >= 190)
        cout << "\nEs alto/a";
    else
        cout << "\nNo es alto/a";
}
else{                                     // <- edad < 18
    cout << "\nEs menor de edad";

    if (altura >= 175)
        cout << "\nEs alto/a";
    else
        cout << "\nNo es alto/a";
}
```



Ejemplo. Retomemos el ejemplo de la subida salarial de la página 153. La subida por número de hijos se aplicaba a todo el mundo:

```
edad >= 45 && salario_base < 1300  ->  +4%
Otro caso                           ->  +1%

numero_hijos > 2                    ->  +2%
```

Ahora cambiamos el criterio para que la subida por número de hijos **sólo** se aplique en el caso de que se haya aplicado la primera subida (por la edad y salario)

```
edad >= 45 && salario_base < 1300  ->  +4%
    numero_hijos > 2                ->  +2%
Otro caso                           ->  +1%
```

Nos quedaría:

```
.....
salario_final = salario_base;

if (edad >= 45 && salario_base < 1300){
    salario_final = salario_final * 1.04;

    if (numero_hijos > 2)
        salario_final = salario_final * 1.02;
}
else
    salario_final = salario_final * 1.01;
```



198

Ejemplo. Refinamos el programa para el cálculo de las raíces de una ecuación de segundo grado visto en la página 138 para contemplar los casos especiales.

```
.....
if (a != 0) {
    denominador = 2*a;
    radicando = b*b - 4*a*c;

    if (radicando == 0){
        raiz1 = -b / denominador;
        cout << "\nSólo hay una raíz doble: " << raiz1;
    }
    else{
        if (radicando > 0){
            radical = sqrt(radicando);
            raiz1 = (-b + radical) / denominador;
            raiz2 = (-b - radical) / denominador;
            cout << "\nLas raíces son" << raiz1 << " y " << raiz2;
        }
        else
            cout << "\nNo hay raíces reales.";
    }
}
else{
    if (b != 0){
        raiz1 = -c / b;
        cout << "\nEs una recta. La única raíz es " << raiz1;
    }
    else
        cout << "\nNo es una ecuación.";
}
```



Ejercicio. Lea sobre una variable `nota_escrito` la nota de que haya sacado un alumno de FP en el examen escrito. Súbale 0.5 puntos, siempre que haya sacado más de un 4.5. Debe controlar que la nota final no sea mayor de 10, de forma que todos los que hayan sacado entre un 9.5 y un 10, su nota final será de 10.

II.1.4.2. Anidar o no anidar: he ahí el dilema

Nos preguntamos si, en general, es mejor anidar o no. La respuesta es que depende de la situación.

Un ejemplo en el que es mejor no anidar

Ejemplo. Retomemos el ejemplo de la subida salarial de la página 161. Simplificamos el criterio: la subida de sueldo se hará cuando el trabajador tenga una edad mayor o igual de 45 y un salario por debajo de 1300. En otro caso se sube un 1%

```
edad >= 45 && salario_base < 1300  ->  +4%
En otro caso                        ->  +1%
```

.....

```
salario_final = salario_base;
```

```
if (edad >= 45 && salario_base < 1300)
    salario_final = salario_final * 1.04;
else
    salario_final = salario_final * 1.01;
```



Pero observe que también podríamos haber puesto lo siguiente:

```
if (edad >= 45)
    if (salario_base < 1300)
        salario_final = salario_final * 1.04;
    else
        salario_final = salario_final * 1.01;
else
    salario_final = salario_final * 1.01;
```



En general, las siguientes estructuras son equivalentes:

<code>if (c1 && c2)</code>	<code>if (c1)</code>
<code>bloque_A</code>	<code>if (c2)</code>
<code>else</code>	<code>bloque_A</code>
<code>bloque_B</code>	<code>else</code>
	<code>bloque_B</code>
	<code>else</code>
	<code>bloque_B</code>

Para demostrarlo, basta ver bajo qué condiciones se ejecutan los distintos bloques y comprobamos que son las mismas:

Primer caso: bloque_A: c1 true
 c2 true
 bloque_B: (c1&& c2) false:
 c1 true y c2 false
 c1 false y c2 true
 c1 false y c2 false

Segundo caso: bloque_A: c1 true
 c2 true
 bloque_B: c1 true y c2 false
 o bien c1 false

Son equivalentes. Elegimos la primera opción porque la segunda repite código (bloque_B)

Un ejemplo en el que es mejor anidar

Ejemplo. Retomemos el ejemplo de la subida salarial de la página 161. Cambiamos el criterio de subida por el siguiente: todo sigue igual salvo que las subidas de sueldo se harán sólo cuando el trabajador tenga una experiencia de más de dos años. En caso contrario, se le baja el salario un 1%

```
experiencia > 2
    edad >= 45 && salario_base < 1300  ->  +4%
        numero_hijos > 2                ->  +2%
    En otro caso                        ->  +1%
En otro caso                          ->  -1%
```

.....

```
salario_final = salario_base;
```

```
if (experiencia > 2){
    if (edad >= 45 && salario_base < 1300){
        salario_final = salario_final * 1.04;

        if (numero_hijos > 2)
            salario_final = salario_final * 1.02;
    }
    else
        salario_final = salario_final * 1.01;
}
else
    salario_final = salario_final * 0.99;
```



198

http://decsai.ugr.es/jccubero/FP/II_actualizacion_salarial_con_literales.cpp

Si construimos las negaciones correspondientes al `else` de cada condición, el código anterior es equivalente al siguiente:

```
.....
salario_final = salario_base;

if (experiencia > 2 && edad >= 45 && salario_base < 1300)
    salario_final = salario_final * 1.04;
if (experiencia > 2 && edad >= 45
    && salario_base < 1300 && numero_hijos > 2)
    salario_final = salario_final * 1.02;
if (experiencia > 2 && (edad < 45 || salario_base >= 1300))
    salario_final = salario_final * 1.01;
if (experiencia <= 2)
    salario_final = salario_final * 0.99;
```



Obviamente, el anterior código es nefasto ya que no cumple el principio de una única vez.

<pre>if (c1 && c2 && c3) <accion_A> if (c1 && c2 && !c3) <accion_B> if (c1 && !c2) <accion_C> if (!c1) <accion_D></pre>	→	<pre>if (c1) if (c2) if (c3) <accion_A> else <accion_B> else <accion_C> else <accion_D></pre>
---	---	---



Podemos resumir lo visto en el siguiente consejo (que es una generalización del visto en la página 147)

Usaremos los condicionales dobles y anidados de forma coherente para no duplicar ni el código de las sentencias ni el de las expresiones lógicas de los condicionales, cumpliendo así el principio de una única vez.

Otro ejemplo en el que es mejor anidar

Ejemplo. Leemos dos enteros y presentamos un menú de operaciones al usuario.

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

int main(){
    double dato1, dato2, resultado;
    char opcion;

    cout << "\nIntroduce el primer operando: ";
    cin >> dato1;
    cout << "\nIntroduce el segundo operando: ";
    cin >> dato2;
    cout << "\nElija (S)Sumar, (R)Restar, (M)Multiplicar: ";
    cin >> opcion;
    opcion = toupper(opcion);

    if (opcion == 'S')
        resultado = dato1 + dato2;
    if (opcion == 'R')
        resultado = dato1 - dato2;
    if (opcion == 'M')
        resultado = dato1 * dato2;
    if (opcion != 'R' && opcion != 'S' && opcion != 'M')
        resultado = NAN;    // <- Hay que incluir cmath

    cout << "\nResultado = " << resultado;
}
```



Las condiciones `opcion == 'S'`, `opcion == 'R'`, etc, son mutuamente excluyentes entre sí. Cuando una sea `true` las otras serán `false`, pero estamos obligando al compilador a evaluar innecesariamente dichas condiciones.

Para resolverlo usamos estructuras condicionales dobles anidadas:

```
if (opcion == 'S')
    resultado = dato1 + dato2;
else
    if (opcion == 'R')
        resultado = dato1 - dato2;
    else
        if (opcion == 'M')
            resultado = dato1 * dato2;
        else
            resultado = NAN;
```

Una forma también válida de tabular es la siguiente:

```
if (opcion == 'S')
    resultado = dato1 + dato2;
else if (opcion == 'R')
    resultado = dato1 - dato2;
else if (opcion == 'M')
    resultado = dato1 * dato2;
else
    resultado = NAN;    // <- Hay que incluir cmath
```



http://decsai.ugr.es/jccubero/FP/II_menu_operaciones.cpp

También podríamos haber asociado las opciones a los caracteres
'+', '-', '*'

Si debemos comprobar varias condiciones, todas ellas mutuamente excluyentes entre sí, usaremos estructuras condicionales dobles anidadas

Cuando c_1 , c_2 y c_3 sean mutuamente excluyentes:

if (c1)		if (c1)
...		...
if (c2)		else if (c2)
...	→	...
if (c3)		else if (c3)
...		...



II.1.5. Estructura condicional múltiple

Retomemos el ejemplo de lectura de dos enteros y una opción de la página 170. Cuando tenemos que comprobar condiciones mutuamente excluyentes sobre un dato entero, podemos usar otra estructura alternativa.

```
switch (<expresión>) {  
    case <constante1>:  
        <sentencias1>  
        break;  
    case <constante2>:  
        <sentencias2>  
        break;  
    .....  
    [default:  
        <sentencias>]  
}
```

- ▷ <expresión> es un expresión entera.
- ▷ <constante i> es un literal entero.
- ▷ switch sólo comprueba la igualdad.
- ▷ No debe haber dos cases con la misma <constante> en el mismo switch. Si esto ocurre, sólo se ejecutan las sentencias del caso que aparezca primero.
- ▷ El identificador especial default permite incluir un caso por defecto, que se ejecutará si no se cumple ningún otro. Lo pondremos al final de la estructura como el último de los casos.

```
if (opcion == 'S')
    resultado = dato1 + dato2;
else if (opcion == 'R')
    resultado = dato1 - dato2;
else if (opcion == 'M')
    resultado = dato1 * dato2;
else{
    resultado = NAN;
}
```

es equivalente a:

```
switch (opcion){
    case 'S':
        resultado = dato1 + dato2;
        break;
    case 'R':
        resultado = dato1 - dato2;
        break;
    case 'M':
        resultado = dato1 * dato2;
        break;
    default:
        resultado = NAN;
}
```


El gran problema con la estructura `switch` es que el programador olvidará en más de una ocasión, incluir la sentencia `break`. La única *ventaja* es que se pueden realizar las mismas operaciones para varias constantes:

```
cin >> opcion;

switch (opcion){
    case 'S':
    case 's':
        resultado = dato1 + dato2;
        break;
    case 'R':
    case 'r':
        resultado = dato1 - dato2;
        break;
    case 'M':
    case 'm':
        resultado = dato1 * dato2;
        break;
    default:
        resultado = NAN;
}
```

El compilador no comprueba que cada case termina en un break. Es por ello, que debemos evitar, en la medida de lo posible, la sentencia switch y usar condicionales anidados en su lugar.

II.1.6. Ámbito de un dato (revisión)

En la página 39 se introdujo el concepto de ámbito de un dato como los sitios en los que se conoce un dato.

Hasta ahora, los datos los hemos declarado al inicio del programa. Vamos a ver que esto no es una restricción del lenguaje. De hecho, podemos declarar datos prácticamente en cualquier lugar: por ejemplo, dentro de un bloque condicional. Su ámbito será únicamente dicho bloque, es decir, el dato no se conocerá fuera del bloque.

Ejemplo. Retomemos el ejemplo de la página 138. Podemos declarar las variables `radical` y `denominador` dentro del condicional, siempre que no se necesiten fuera de él.

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

int main(){
    double a, b, c;           // Parámetros de la ecuación
    double raiz1, raiz2;      // Raíces obtenidas

    cout << "\nIntroduce coeficiente de 2º grado: ";
    cin >> a;
    cout << "\nIntroduce coeficiente de 1er grado: ";
    cin >> b;
    cout << "\nIntroduce coeficiente independiente: ";
    cin >> c;

    if (a != 0) {
        double radical, denominador;

        denominador = 2*a;
```



210

```
    radical = sqrt(b*b - 4*a*c);

    raiz1 = (-b + radical) / denominador;
    raiz2 = (-b - radical) / denominador;

    cout << "\nLas raíces son" << raiz1 << " y " << raiz2;
}
else{
    raiz1 = -c/b;
    cout << "\nLa única raíz es " << raiz1;
    // cout << radical;  <- Error de compilación
}

// cout << denominador;  <- Error de compilación
}
```

Por ahora, no abusaremos de la declaración de datos dentro de un bloque condicional. Lo haremos sólo cuando sea muy evidente que los datos declarados no se necesitarán fuera del bloque.

II.1.7. Algunas cuestiones sobre condicionales

II.1.7.1. Cuidado con la comparación entre reales

La expresión $1.0 == (1.0/3.0)*3.0$ podría evaluarse a false debido a la precisión finita para calcular $(1.0/3.0)$ (0.333333333)

Otro ejemplo:

```
double raiz_de_dos;

raiz_de_dos = sqrt(2.0);

if (raiz_de_dos * raiz_de_dos != 2.0)    // Podría evaluarse a true
    cout << ";Raíz de dos al cuadrado no es igual a dos!";
```

Otro ejemplo:

```
double descuento_base, porcentaje;

descuento_base = 0.1;
porcentaje = descuento_base * 100;

if (porcentaje == 10.0)    // Podría evaluarse a false :-0
    cout << "Descuento del 10%";
```

Recuerde que la representación en coma flotante no es precisa, por lo que 0.1 será internamente un valor próximo a 0.1, pero no igual.

Soluciones:

- ▷ **Fomentar condiciones de desigualdad cuando sea posible.**
- ▷ **Fijar un *error* de precisión y aceptar igualdad cuando la diferencia de las cantidades sea menor que dicho error.**

```
double real1, real2;  
const double epsilon = 0.00001;
```

```
if (real1 - real2 < epsilon)  
    cout << "Son iguales";
```

Para una solución aún mejor consulte:

[https://randomascii.wordpress.com/2012/02/25/
comparing-floating-point-numbers-2012-edition/](https://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/)

II.1.7.2. Evaluación en ciclo corto y en ciclo largo

En una condición compuesta del tipo `A && B && C`, si la expresión `A` fuese `false`, ya no sería necesario evaluar ni `B`, ni `C`. Lo mismo ocurriría si la expresión fuese del tipo `A || B || C` y `A` fuese `true`. Los compiladores pueden analizar a priori una expresión compuesta y dejar de evaluar las expresiones que la constituyen en cuanto ya no sea necesario.

Evaluación en ciclo corto (Short-circuit evaluation) : El compilador optimiza la evaluación de expresiones lógicas evaluando sus términos de izquierda a derecha hasta que ya no sea necesario. La mayoría de los compiladores realizan este tipo de evaluación por defecto.

Evaluación en ciclo largo (Eager evaluation) : El compilador evalúa todos los términos de la expresión lógica para conocer el resultado de la expresión completa.

Ejemplo.

```
if (n != 0 && total/n < tope) ...
```

Supongamos que `n` es cero.

- ▷ **Ciclo largo**: El compilador evalúa (innecesariamente) todas las expresiones lógicas. En este caso, se podría producir un error al dividir por cero.
- ▷ **Ciclo corto**: El compilador evalúa sólo la primera expresión lógica.

Ejercicio. Re-escriba el anterior ejemplo (evitando dividir por cero) usando otra estructura que no involucre una condición compuesta. Para resolverlo, observe que la condición `total/n < tope` es dependiente de la condición `n != 0`, en el sentido de que si la segunda es `false`, no queremos evaluar la otra.

II.1.8. Programando como profesionales

II.1.8.1. Diseño de algoritmos fácilmente extensibles

Ejemplo. Calcule el mayor de tres números a, b, c

La primera idea podría ser contemplar todas las posibilidades de orden:

$$b \leq a \leq c, a \leq b \leq c, a \leq c \leq b, \dots$$

y construir un condicional `if` para cada una de ellas. Esto generaría demasiadas expresiones lógicas:

Algoritmo: Mayor de tres números. Versión 1

▷ **Idea:** Analizando en qué situación(es) el máximo es a, b o c .

▷ **Entradas:** a, b y c

Salidas: El mayor entre a, b y c

▷ **Descripción:**

Si a es mayor que los otros, el mayor es a

Si b es mayor que los otros, el mayor es b

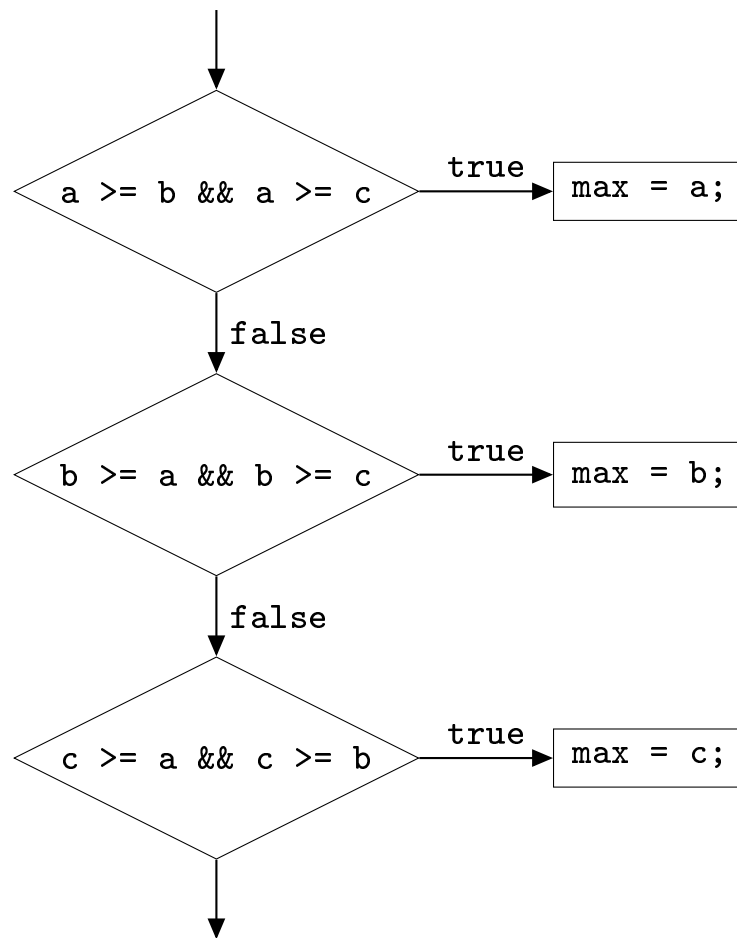
Si c es mayor que los otros, el mayor es c

▷ **Implementación:**

```
if ((a >= b) && (a >= c))
    max = a;
if ((b >= a) && (b >= c))
    max = b;
if ((c >= a) && (c >= b))
    max = c;
```



187



Inconvenientes:

- ▷ Al tener tres condicionales seguidos, siempre se evalúan las 6 expresiones lógicas. En cuanto hallemos el máximo deberíamos parar y no seguir preguntando.
- ▷ Podemos resolver el problema usando menos de 6 expresiones lógicas.

Ejemplo. El mayor de tres números (segunda aproximación)

Algoritmo: Mayor de tres números. Versión 2

- ▷ **Idea:** Ir descartando conforme se van comparando
- ▷ **Entradas y Salidas:** idem
- ▷ **Descripción:**

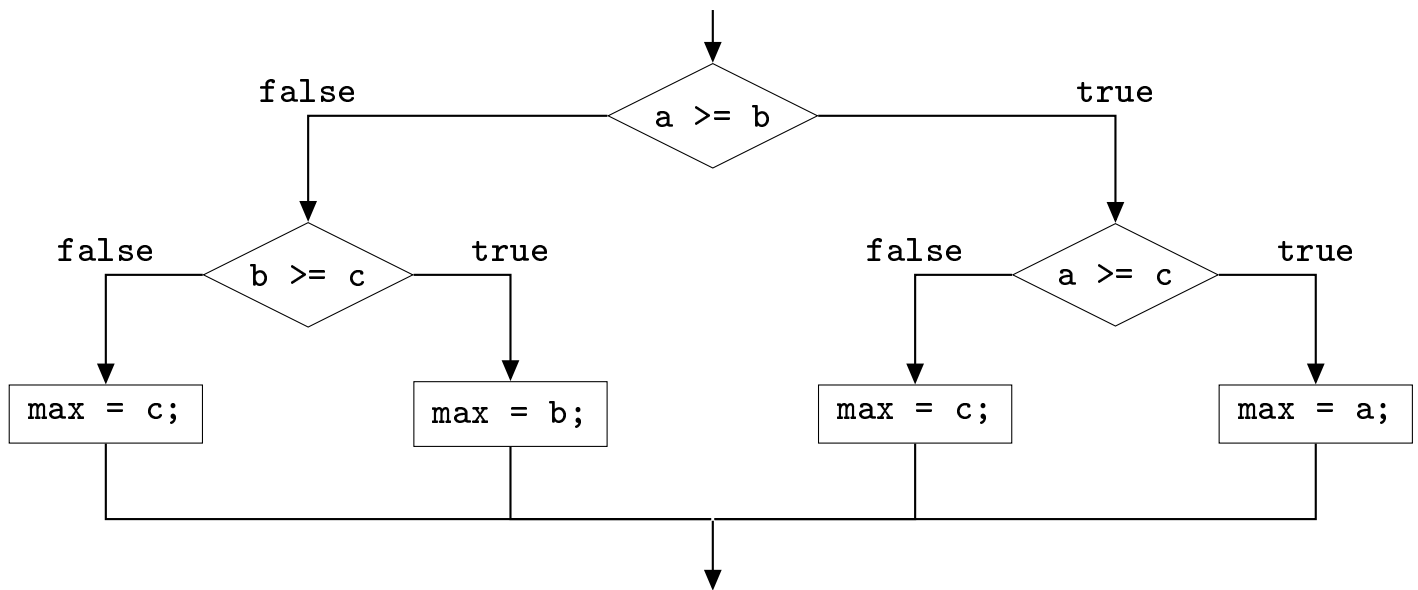
```
Si a es mayor que b, entonces
    Calcular el máximo entre a y c
En otro caso,
    Calcular el máximo entre b y c
```

- ▷ **Implementación:**

```
if (a >= b)
    if (a >= c)
        max = a;
    else
        max = c;
else
    if (b >= c)
        max = b;
    else
        max = c;
```



187



Inconvenientes:

- ▷ Repetimos código: `max = c;`
- ▷ La solución es difícil de extender a más valores. Observe cómo quedaría el mayor de cuatro números con esta aproximación:

```
if (a >= b)
    if (a >= c)
        if (a >= d)
            max = a;
        else
            max = d;
    else
        if (c >= d)
            max = c;
        else
            max = d;
else
    if (b >= c)
        if (b >= d)
            max = b;
        else
            max = d;
    else
        if (c >= d)
            max = c;
        else
            max = d;
```



187



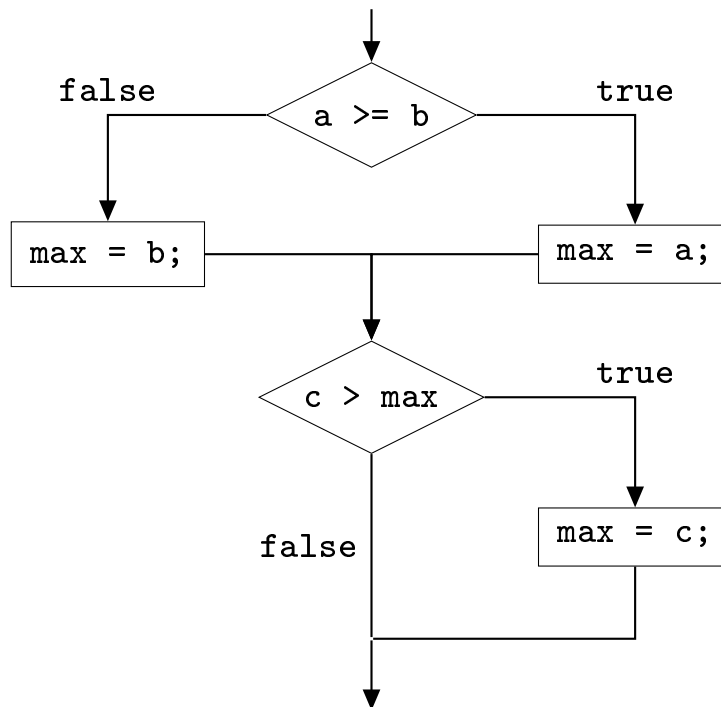
Ejemplo. El mayor de tres números (tercera aproximación)

Algoritmo: Mayor de tres números. Versión 3

- ▷ **Idea:** Ir actualizando el máximo.
- ▷ **Entradas y Salidas:** idem
- ▷ **Descripción e Implementación:**

```
/*  
Calcular el máximo (max) entre a y b.  
Calcular el máximo entre max y c.  
*/  
  
if (a >= b)  
    max = a;  
else  
    max = b;  
  
if (c > max)  
    max = c;
```

Observe que con el primer `if` garantizamos que la variable `max` tiene un valor (o bien `a` o bien `b`). Por tanto, en el último condicional basta usar una desigualdad estricta.



Ventajas:

- ▷ Es mucho más fácil de entender
- ▷ No repite código
- ▷ Es mucho más fácil de extender a varios valores. Veamos cómo quedaría con cuatro valores:

Algoritmo: Mayor de cuatro números

- ▷ **Entradas y Salidas:** idem
- ▷ **Descripción e Implementación:**

```
/*  
Calcular el máximo (max) entre a y b.  
Calcular el máximo entre max y c.  
Calcular el máximo entre max y d.  
*/  
  
if (a >= b)  
    max = a;  
else  
    max = b;  
  
if (c > max)  
    max = c;  
  
if (d > max)  
    max = d;
```



http://decsai.ugr.es/jccubero/FP/II_max.cpp

En general:

```
Calcular el máximo (max) entre a y b.  
Para cada uno de los valores restantes,  
    max = máximo entre max y el nuevo valor.
```

**Diseña los algoritmos para que sean fácilmente
extensibles a situaciones más generales.**



Otra alternativa sería inicializar `max` al primer valor e ir comparando con el resto:

```
max = a;

if (b > max)
    max = b;

if (c > max)
    max = c;

if (d > max)
    max = d;
```

En general:

```
Inicializar el máximo a un valor cualquiera -a-
Para cada uno de los valores restantes,
    max = máximo entre max y el nuevo valor.
```

Algunas citas sobre la importancia de escribir código que sea fácil de entender:

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand".

Martin Fowler



"Programs must be written for people to read, and only incidentally for machines to execute".

Abelson & Sussman



"Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live".



Un principio de programación transversal:

Principio de Programación:

Sencillez (Simplicity)

Fomente siempre la sencillez y la legibilidad en la escritura de código



"There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."

C.A.R. Hoare



II.1.8.2. Descripción de un algoritmo

Se trata de describir la idea principal del algoritmo, de forma concisa y esquemática, sin entrar en detalles innecesarios.

```
if (a >= b)
    max = a;
else
    max = b;

if (c > max)
    max = c;
```

Una lamentable descripción del algoritmo:

Compruebo si $a \geq b$. En ese caso, lo que hago es asignarle a la variable max el valor a y si no le asigno el otro valor, b. Una vez hecho esto, paso a comprobar si el otro valor, es decir, c, es mayor que max (la variable anterior), en cuyo caso le asigno a max el valor c y si no, no hago nada.



El colmo:

Compruevo si $a \geq b$ y en ese caso lo que ago es asignarle a la variable max el valor a y sino le asigno el otro valor b una vez echo esto paso a comprobar si el otro valor es decir c es mayor que max (la variable anterior) en cuyo caso le asigno a max el valor c y sino no ago nada



▷ ***Nunca parafrasearemos el código***

```
/* Si a es >= b, asignamos a max el valor a
   En otro caso, le asignamos b.
   Una vez hecho lo anterior, vemos si
   c es mayor que max, en cuyo caso
   le asignamos c
*/
```



```
if (a >= b)
    max = a;
else
    max = b;

if (c > max)
    max = c;
```

▷ ***Seremos esquemáticos (pocas palabras en cada línea)***

```
/* Calcular el máximo entre a y b, y una vez hecho
   esto, pasamos a calcular el máximo entre el anterior,
   al que llamaremos max, y el nuevo valor c
*/
```



```
/* Calcular el máximo (max) entre a y b.
   Calcular el máximo entre max y c.
*/
if (a >= b)
    max = a;
else
    max = b;

if (c > max)
    max = c;
```



▷ **Comentaremos un bloque completo**

La descripción del algoritmo la incluiremos antes de un bloque, pero nunca entre las líneas del código. Esto nos permite separar las dos partes y poder leerlas independientemente.

```
// Calcular el máximo entre a y b
if (a >= b)
    max = a;
else
    // En otro caso:
    max = b;
// Calcular el máximo entre max y c
if (c > max)
    max = c;
```



Algunos autores incluyen la descripción a la derecha del código, pero es mucho más difícil de mantener ya que si incluimos líneas de código nuevas, se *descompone* todo el comentario:

```
if (a >= b)          // Calcular el máximo
    max = a;         // entre a y b
else
    max = b;

if (c > max)          // Calcular el máximo
    max = c;         // entre max y c
```



Use descripciones de algoritmos que sean CONCISAS con una presentación visual agradable y esquemática.

**No se hará ningún comentario de aquello que sea obvio.
Más vale poco y bueno que mucho y malo.**

Las descripciones serán de un BLOQUE completo.

Sólo se usarán comentarios al final de una línea en casos puntuales, para aclarar el código de dicha línea.



II.1.8.3. Descomposición de una solución en tareas

Si observamos que una misma operación se repite en todas las partes de una estructura condicional, debemos ver si es posible su extracción fuera de la estructura.

Ejemplo. Una empresa aplica la siguiente promoción en la venta de sus artículos: si el número de unidades pedidas es mayor de 50, se aplicará un descuento del 3 %. Lea el precio de venta por unidad, la cantidad de artículos pedida y calcule el precio final, aplicando un IVA del 16 %.

```
cin >> pvp_unidad >> unidades_vendidas;

if (unidades_vendidas < 50){
    pvp = pvp_unidad * unidades_vendidas;
    pvp = pvp * 1.16;
}
else{
    pvp = pvp_unidad * unidades_vendidas;
    pvp = pvp * 0.97;
    pvp = pvp * 1.16;
}
```



La sentencia `pvp = pvp_unidad * unidades_vendidas;` **se repite en ambas partes del condicional. Debemos sacarlo de éste y realizar la operación antes de entrar al if. La aplicación del IVA también se repite, por lo que debemos sacar dicha operación y realizarla después del condicional:**

```
cin >> pvp_unidad >> unidades_vendidas;
pvp = pvp_unidad * unidades_vendidas;

if (unidades_vendidas >= 50)
    pvp = pvp * 0.97;
pvp = pvp * 1.16;
```

Finalmente, como ya sabemos, no debemos utilizar *números mágicos* (vea la página 35) en el código sino constantes. Nos quedaría:

```
const int MIN_UNID_DSCTO = 50;
const double IV_DSCTO = 1 - 0.05;
const double IV_IVA = 1 + 0.16;
.....
cin >> pvp_unidad >> unidades_vendidas;

pvp = pvp_unidad * unidades_vendidas;

if (unidades_vendidas >= MIN_UNID_DSCTO)
    pvp = pvp * IV_DSCTO;

pvp = pvp * IV_IVA;
```



http://decsai.ugr.es/jccubero/FP/II_UnidadesVendidas.cpp

Lo que hemos hecho ha sido separar las tareas principales de este problema, a saber:

- ▷ Calcular el precio de todas las unidades vendidas
- ▷ Aplicar el descuento, en su caso
- ▷ Aplicar el IVA

Son tres tareas independientes que estaban duplicadas en la primera versión. A veces, la duplicación del código que se produce al repetir la resolución de una misma tarea en varios sitios no es tan fácil de identificar. Lo vemos en el siguiente ejemplo.

Ejemplo. Retomemos el ejemplo de la subida salarial de la página 166.

```
.....
salario_final = salario_base;

if (experiencia > 2) {
    if (edad >= 45 && salario_base < 1300){
        salario_final = salario_final * 1.04;

        if (numero_hijos > 2)
            salario_final = salario_final * 1.02;
    }
    else
        salario_final = salario_final * 1.01;
}
else
    salario_final = salario_final * 0.99;
```

Podemos resolver el problema descomponiendo la tarea principal en dos sub-tareas:

- 1. Calcular el porcentaje de actualización**
- 2. Aplicar dicho porcentaje**

```
double IV_salario;      // Índice de variación
.....
salario_final = salario_base;

// Calcular el porcentaje de actualización

if (experiencia > 2) {
    if (edad >= 45 && salario_base < 1300){
        IV_salario = 1.04;

        if (numero_hijos > 2)      // Observe esta asignación:
            IV_salario = IV_salario * 1.02;
    }
    else
        IV_salario = 1.01;
}
else
    IV_salario = 0.99;

// Aplicar dicho porcentaje

salario_final = salario_final * IV_salario;
```

Observe la actualización del 2%. Ahora queda mucho más claro que la subida del 2% es sobre la subida anterior (la del 4%)

Analice siempre con cuidado las tareas a resolver en un problema e intente separar los bloques de código que resuelven cada una de ellas.

IMPORTANT

Si tenemos previsto utilizar los límites de las subidas salariales y los porcentajes correspondientes en otros sitios del programa, debemos usar constantes en vez de literales:

```
const int MINIMO_EXPERIENCIA_ALTA = 2,
        MINIMO_FAMILIA_NUMEROSA = 2, ...

.....
const double IV_SENIOR_Y_SAL_BAJO      = 1.04,
            IV_NO_SENIOR_Y_SAL_BAJO = 1.01, ...

.....
es_experiencia_alta = experiencia > MINIMO_EXPERIENCIA_ALTA;
es_familia_numerosa = numero_hijos > MINIMO_FAMILIA_NUMEROSA;
es_salario_bajo      = salario_base < MAXIMO_SALARIO_BAJO;
es_edad_senior       = edad >= MINIMO_EDAD_SENIOR;

if (es_experiencia_alta){
    if (es_edad_senior && es_salario_bajo){
        IV_salario = IV_SENIOR_Y_SAL_BAJO;

        if (es_familia_numerosa)
            IV_salario = IV_salario * IV_FAMILIA_NUMEROSA;
    }
    else
        IV_salario = IV_NO_SENIOR_Y_SAL_BAJO;
}
else
    IV_salario = IV_SIN_EXPERIENCIA;

salario_final = salario_final * IV_salario;
.....
```



http://decsai.ugr.es/jccubero/FP/II_actualizacion_salarial.cpp

II.1.8.4. Las expresiones lógicas y el principio de una única vez

Según el principio de una única vez (página 112) no debemos repetir el código de una expresión en distintas partes del programa, si la evaluación de ésta no varía. Lo mismo se aplica si es una expresión lógica.

Ejemplo. Retomamos el ejemplo de subir la nota de la página 163. Imprimimos un mensaje específico si ha superado el examen escrito:

```
double nota_escrito;
.....
if (nota_escrito >= 4.5){
    nota_escrito = nota_escrito + 0.5;

    if (nota_escrito > 10)
        nota_escrito = 10;
}
.....
if (nota_escrito >= 4.5)                // <- repite código 😞
    cout << "Examen escrito superado con la nota: " << nota_escrito;
.....
```

Si ahora quisiéramos cambiar cambiamos el criterio a que sea mayor estricto, tendríamos que modificar el código de dos líneas. Para resolver este problema, introducimos una variable lógica:

```
double nota_escrito;
bool escrito_superado;
.....
escrito_superado = nota_escrito >= 4.5;

if (escrito_superado){
    nota_escrito = nota_escrito + 0.5;

    if (nota_escrito > 10)
        nota_escrito = 10;
}
.....
if (escrito_superado)
    cout << "Examen escrito superado con la nota: " << nota_escrito;
.....
```



Nota:

Como siempre, deberíamos usar constantes

```
const double NOTA_MINIMA_APROBAR = 4.5;
const double MAX_NOTA = 10.0;
const double SUBIDA_NOTA_APROBADOS = 0.5;
```

en vez de los literales 4.5, 10, 0.5 en las sentencias del programa.

En resumen:

El uso de variables intermedias para determinar criterios nos ayuda a no repetir código y facilita la legibilidad de éste. Se les asigna un valor en un bloque y se observa su contenido en otro.

II.1.8.5. Separación de entradas/salidas y cálculos

En temas posteriores se introducirán herramientas para aislar bloques de código en módulos (clases, funciones, etc). Será importante que los módulos que hagan entradas y salidas de datos no realicen también otro tipo de cálculos ya que, de esta forma:

- ▷ Se separan responsabilidades.

Cada módulo puede actualizarse de forma independiente.

- ▷ Se favorece la reutilización entre plataformas (Linux, Windows, etc).

Las E/S en un entorno de ventanas como Windows no se hacen como se hacen en modo consola (con `cout`). Por lo tanto, el código que se encarga de esta parte será distinto. Pero la parte del código que se encarga de realizar los cálculos será el mismo: si dicho código está *empaquetado* en un módulo, éste podrá reutilizarse tanto en un programa que interactúe con la consola como en un programa que funcione en un entorno de ventanas.

Por ahora, trabajamos en un único fichero y sin módulos (éstos se introducen en el tema IV). Pero al menos, perseguiremos el objetivo de separar E/S y C, separando los bloques de código de cada parte.

La siguiente norma es un caso particular de la indicada en la página 197

Los bloques de código que realizan entradas o salidas de datos (`cin`, `cout`) estarán separados de los bloques que realizan cálculos.

IMPORTANT

El uso de variables intermedias nos ayuda a separar los bloques de E/S y C. Se les asigna un valor en un bloque y se observa su contenido en otro.

En la mayor parte de los ejemplos vistos en este tema hemos respetado esta separación. Veamos algunos ejemplos de lo que no debemos hacer.

Ejemplo. Retomamos los ejemplos de la página 139. Para no mezclar E/S y C, debemos sustituir el código siguiente:

```
// Cómputos mezclados con la salida de resultados: 😞  
  
if (entero % 2 == 0)  
    cout << "\nEs par";  
else  
    cout << "\nEs impar";  
  
cout << "\nFin del programa";
```

por:

```
bool es_par;  
.....  
// Cómputos: 😊  
  
es_par = entero % 2 == 0;  
  
// Salida de resultados:  
  
if (es_par)  
    cout << "\nEs par";  
else  
    cout << "\nEs impar";  
  
cout << "\nFin del programa";
```

Nos preguntamos si podríamos usar una variable de tipo `string` en vez de un `bool`:

```
string tipo_de_entero;
.....
// Cómputos:

if (entero % 2 == 0)
    tipo_de_entero = "es par"
else
    tipo_de_entero = "es impar";

// Salida de resultados:

if (tipo_de_entero == "es_par")
    cout << "\nEs par";
else
    cout << "\nEs impar";

cout << "\nFin del programa";
```



¿Localiza el error?

Estamos comparando con una cadena distinta de la asignada (cambia un único carácter)

Jamás usaremos un tipo `string` para detectar un número limitados de alternativas posibles ya que es propenso a errores.

Ejemplo. Retomamos el ejemplo del máximo de tres valores de la página 185

```
// Cómputos: Calculamos max

if (a >= b)
    max = a;
else
    max = b;

if (c > max)
    max = c;

// Salida de resultados: Observamos el valor de max

cout << "\nMáximo: " << max;
```



El siguiente código mezcla E/S y C dentro del mismo bloque condicional:

```
// Calculamos el máximo Y lo imprimimos en pantalla.
// Mezclamos E/S con C

if (a >= b)
    cout << "\nMáximo: " << a;
else
    cout << "\nMáximo: " << b;

if (c > max)
    cout << "\nMáximo: " << c;
```



Observe la similitud con lo visto en la página 196. En esta segunda versión del máximo no se han separado las tareas de calcular el máximo e imprimir el resultado.

Ejemplo. Retomamos el ejemplo de la edad y altura de una persona de la página 160. Para separar E/S y Cómputos, introducimos las variables intermedias `es_alto`, `es_mayor_edad`. Además, usamos constantes en vez de literales para representar y operar con los umbrales.

```
int main(){
    const int MAYORIA_EDAD = 18,
            UMBRAL_ALTURA_JOVENES = 175,
            UMBRAL_ALTURA_ADULTOS = 190;
    int edad, altura;
    bool es_alto, es_mayor_edad, umbral_altura;

    // Entrada de datos:

    cout << "Introduzca los valores de edad y altura: ";
    cin >> edad;
    cin >> altura;

    // Cómputos:

    es_mayor_edad = edad >= MAYORIA_EDAD;

    if (es_mayor_edad)
        umbral_altura = UMBRAL_ALTURA_ADULTOS;
    else
        umbral_altura = UMBRAL_ALTURA_JOVENES;

    es_alto = altura >= umbral_altura;
```




```
// Salida de resultados:

cout << "\n\n";

if (es_mayor_edad)
    cout << "Es mayor de edad";
else
    cout << "Es menor de edad";

if (es_alto)
    cout << "Es alto/a";
else
    cout << "No es alto/a";
}
```

http://decsai.ugr.es/jccubero/FP/II_altura.cpp

II.1.8.6. El tipo enumerado y los condicionales

Hay situaciones en las que necesitamos manejar información que sólo tiene unos cuantos valores posibles.

- ▷ **Calificación ECTS de un alumno:** {A, B, C, D, E, F, G}
- ▷ **Tipo de letra:** {es mayúscula, es minúscula, es otro carácter}
- ▷ **Puesto alcanzado en la competición:** {primero, segundo, tercero}
- ▷ **Día de la semana:** {lunes, ... , domingo}
- ▷ **Categoría laboral de un empleado:** {administrativo, programador, analista, directivo }
- ▷ **Tipo de ecuación de segundo grado:** {una única raíz, ninguna solución, ... }

Opciones con lo que conocemos hasta ahora:

- ▷ **Una variable de tipo `char` o `int`.** El inconveniente es que el código es propenso a errores:

```
char categoria_laboral;

categoria_laboral = 'a';           // Analista
categoria_laboral = 'm';           // adMinistrativo
categoria_laboral = 'x';           // Categoría inexistente
if (categoria_laboral == 'w')      // Categoría inexistente
    .....
```



- ▷ **Un dato `bool` por cada categoría:**

```
bool es_administrativo, es_programador,
     es_analista, es_directivo;
```



Inconvenientes: Debemos manejar cuatro variables por separado y además representan 8 opciones distintas, en vez de cuatro.

Solución: Usar un tipo enumerado. A un dato de tipo *enumerado* (*enumeration*) sólo se le puede asignar un número muy limitado de valores. Éstos son especificados por el programador. Primero se define el *tipo* (lo haremos antes del `main`) y luego la variable de dicho tipo.

Para nombrar los valores del enumerado, usaremos minúsculas. Es un estilo admitido en Google C++ Style Guide, aunque recomiendan ir sustituyéndolo por el uso de mayúsculas (como se hace con las constantes) Nosotros usaremos minúsculas para asemejarlo a los valores `true`, `false` de un `bool`. Usaremos el estilo `UpperCamelCase` para el nombre del tipo de dato.

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

enum class CalificacionECTS
    {A, B, C, D, E, F, G}; // No van entre comillas!
enum class PuestoPodium
    {primero, segundo, tercero};
enum class CategoriaLaboral
    {administrativo, programador, analista, directivo};

int main(){
    CalificacionECTS nota;
    PuestoPodium      podium;
    CategoriaLaboral categoria_laboral;

    nota              = CalificacionECTS::A;
    podium            = PuestoPodium::primero;
    categoria_laboral = CategoriaLaboral::programador;

    // Las siguientes sentencias dan error de compilación
    // categoria_laboral = CategoriaLaboral::peon
```



```
// categoria_laboral = peon;  
// categoria_laboral = 'a';  
// cin >> categoria_laboral;  
.....  
}
```

El tipo enumerado puede verse como una extensión del tipo `bool` ya que nos permite manejar más de dos opciones excluyentes.

Al igual que un `bool`, un dato enumerado contendrá un único valor en cada momento. La diferencia está en que con un `bool` sólo tenemos 2 posibilidades y con un enumerado tenemos más (pero no tantas como las 256 de un `char`, por ejemplo).

¿Qué operaciones se pueden hacer sobre un enumerado? Por ahora, sólo tiene sentido la comparación de igualdad.

Ejemplo. Modifique el código del programa que calcula las raíces de una ecuación de segundo grado (página 162) para separar las E/S de los cálculos. Debemos introducir una variable de tipo enumerado que nos indique el tipo de ecuación (una única raíz doble, dos raíces reales, etc)

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

enum class TipoEcuacion
{
    una_raiz_doble, dos_raices_reales, ninguna_raiz_real,
    recta_con_una_raiz, no_es_ecuacion};

int main(){
    int a, b, c;
    int denominador;
    double radical, radicando, raiz1, raiz2;
    TipoEcuacion tipo_ecuacion;

    // Entrada de datos:

    cout << "\nIntroduce coeficiente de segundo grado: ";
    cin >> a;
    cout << "\nIntroduce coeficiente de 1er grado: ";
    cin >> b;
    cout << "\nIntroduce coeficiente independiente: ";
    cin >> c;

    // Cálculos:

    if (a != 0) {
        denominador = 2*a;
        radicando = b*b - 4*a*c;
```



```
    if (radicando == 0){
        raiz1 = -b / denominador;
        tipo_ecuacion = TipoEcuacion::una_raiz_doble;
    }
    else{
        if (radicando > 0){
            radical = sqrt(radicando);
            raiz1 = (-b + radical) / denominador;
            raiz2 = (-b - radical) / denominador;
            tipo_ecuacion = TipoEcuacion::dos_raices_reales;
        }
        else
            tipo_ecuacion = TipoEcuacion::ninguna_raiz_real;
    }
}
else{
    if (b != 0){
        raiz1 = -c / b;
        tipo_ecuacion = TipoEcuacion::recta_con_una_raiz;
    }
    else
        tipo_ecuacion = TipoEcuacion::no_es_ecuacion;
}

// Salida de Resultados:

cout << "\n\n";

if (tipo_ecuacion == TipoEcuacion::una_raiz_doble)
    cout << "Sólo hay una raíz doble: " << raiz1;
else if (tipo_ecuacion == TipoEcuacion::dos_raices_reales)
    cout << "Las raíces son: " << raiz1 << " y " << raiz2;
else if (tipo_ecuacion == TipoEcuacion::ninguna_raiz_real)
```

```
    cout << "No hay raíces reales.";
else if (tipo_ecuacion == TipoEcuacion::recta_con_una_raiz)
    cout << "Es una recta. La única raíz es: " << raiz1;
else if (tipo_ecuacion == TipoEcuacion::no_es_ecuacion)
    cout << "No es una ecuación.";
```

http://decsai.ugr.es/jccubero/FP/II_ecuacion_segundo_grado.cpp

Si se prefiere, puede usarse una estructura condicional múltiple, pero recordemos que es una estructura a evitar ya que nos obliga a tener que incluir explícitamente un `break` en cada `case`.

```
switch (tipo_ecuacion){
    case TipoEcuacion::una_raiz_doble:
        cout << "Sólo hay una raíz doble: " << raiz1;
        break;
    case TipoEcuacion::dos_raices_reales:
        cout << "Las raíces son: " << raiz1 << " y " << raiz2;
        break;
    case TipoEcuacion::ninguna_raiz_real:
        cout << "No hay raíces reales.";
        break;
    case TipoEcuacion::recta_con_una_raiz:
        cout << "Es una recta. La única raíz es: " << raiz1;
        break;
    case TipoEcuacion::no_es_ecuacion:
        cout << "No es una ecuación.";
        break;
}
```

Ampliación:



Observe que la lectura con `cin` de un enumerado (`cin >> categoria_laboral;`) produce un error en tiempo de ejecución. ¿Cómo leemos entonces los valores de un enumerado desde un dispositivo externo? Habrá que usar una codificación (con caracteres, enteros, etc) y traducirla al enumerado correspondiente.

```
enum class CategoriaLaboral
    {administrativo, programador, analista, directivo};
int main(){
    CategoriaLaboral categoria_laboral;
    char char_categoria_laboral;
    .....
    cin >> char_categoria_laboral;

    if (char_categoria_laboral == 'm')
        categoria_laboral = CategoriaLaboral::administrativo
    else if (char_categoria_laboral == 'p')
        categoria_laboral = CategoriaLaboral::programador
    else if .....
    .....
    if (char_categoria_laboral == CategoriaLaboral::administrativo)
        retencion_fiscal = RETENCION_BAJA;
    else if (char_categoria_laboral == CategoriaLaboral::programador)
        retencion_fiscal = RETENCION_MEDIA;

    salario_netto = salario_bruto -
        salario_bruto * retencion_fiscal/100.0;
```

Una vez leídos los datos y hecha la transformación al enumerado correspondiente, nunca más volveremos a usar los caracteres sino la variable de tipo enumerado.

II.2. Estructuras repetitivas

Una *estructura repetitiva (iteration/loop)* (también conocidas como *bucles, ciclos o lazos*) permite la ejecución de una secuencia de sentencias:

- ▷ o bien, hasta que se satisface una determinada condición → *Bucle controlado por condición (Condition-controlled loop)*
- ▷ o bien, un número determinado de veces → *Bucle controlado por contador (Counter controlled loop)*

II.2.1. Bucles controlados por condición: pre-test y post-test

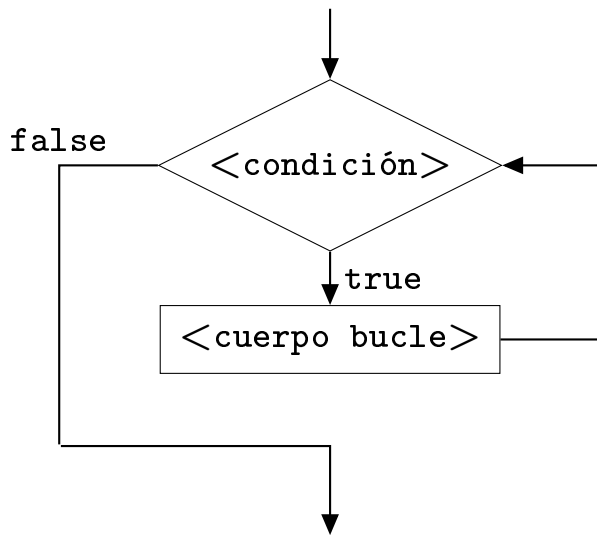
II.2.1.1. Formato

Pre-test	Post-test
<pre>while (<condición>) { <cuerpo bucle> }</pre>	<pre>do{ <cuerpo bucle> }while (<condición>);</pre>

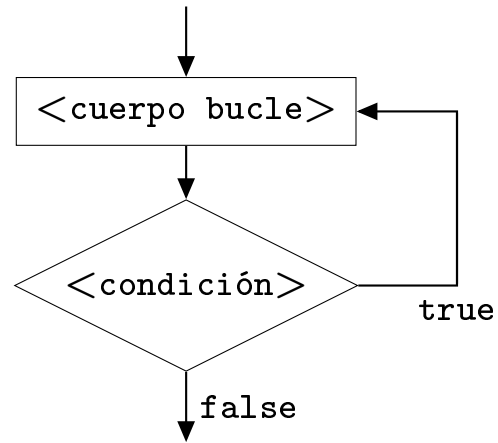
Funcionamiento: En ambos, se va ejecutando el cuerpo del bucle mientras la condición sea verdad.

- ▷ En un *bucle pre-test (pre-test loop)* (`while`) se evalúa la condición antes de entrar al bucle y luego (en su caso) se ejecuta el cuerpo.
- ▷ En un *bucle post-test (post-test loop)* (`do while`) primero se ejecuta el cuerpo y luego se evalúa la condición.

Cada vez que se ejecuta el cuerpo del bucle diremos que se ha producido



(a) while pre test



(b) do-while post test

una *iteración (iteration)*

Al igual que ocurre en la estructura condicional, si los bloques de instrucciones contienen más de una línea, debemos englobarlos entre llaves. En el caso del post-test se recomienda usar el siguiente estilo:

```
do{  
    <cuerpo bucle>  
}while (<condición>);
```



en vez de:

```
do{  
    <cuerpo bucle>  
}  
while (<condición>);
```



ya que, en la segunda versión, si no vemos las instrucciones antes del while, da la impresión que éste es un pre-test.

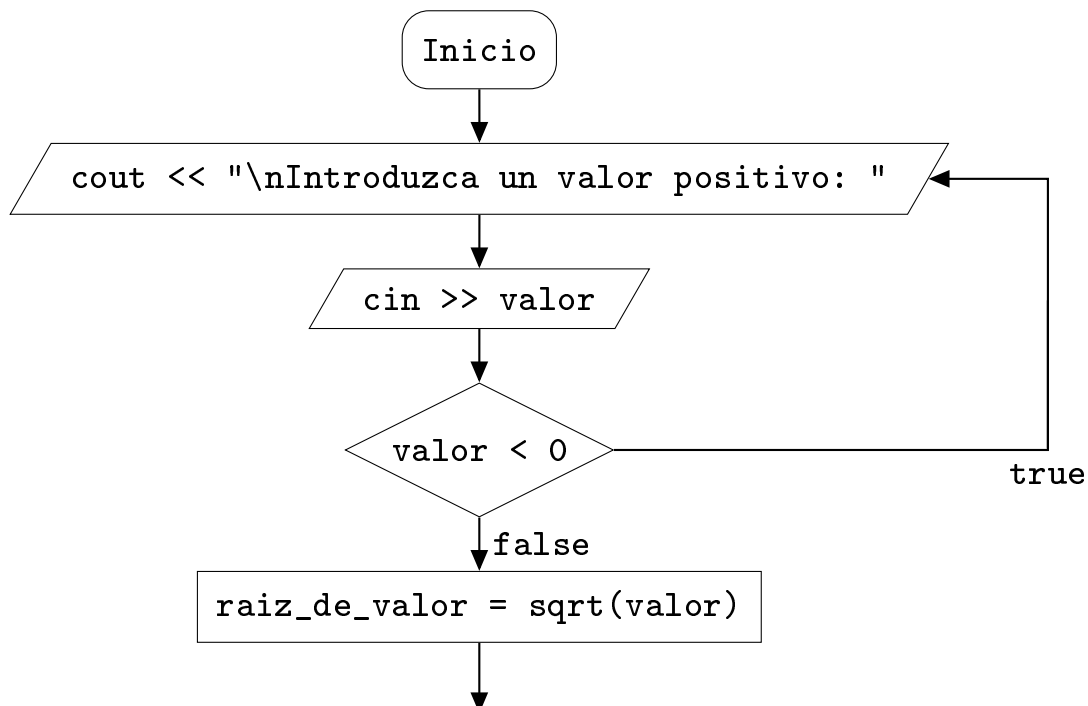
II.2.1.2. Algunos usos de los bucles

Ejemplo. Cree un *filtro (filter)* de entrada de datos: Leer un valor y no permitir al usuario que lo introduzca fuera de un rango determinado. Por ejemplo, que sea un entero positivo para poder calcular la raíz cuadrada:

```
int main(){
    double valor;
    double raiz_de_valor;

    do{
        cout << "\nIntroduzca un valor positivo: ";
        cin >> valor;
    }while (valor < 0);

    raiz_de_valor = sqrt(valor);
    .....
}
```



Nota:

El filtro anterior no nos evita todos los posibles errores. Por ejemplo, si se introduce un valor demasiado grande, se produce un desbordamiento y el resultado almacenado en `valor` es indeterminado.

Nota:

Observe que el estilo de codificación se rige por las mismas normas que las indicadas en la estructura condicional (página 131)

Ejemplo. Escriba 20 líneas con 5 estrellas cada una.

Necesitamos una variable `num_lineas` para contar el número de líneas impresas. Vamos a ver distintas versiones para resolver este problema:

a) Post-test con condición \leq

```
.....
int num_lineas;

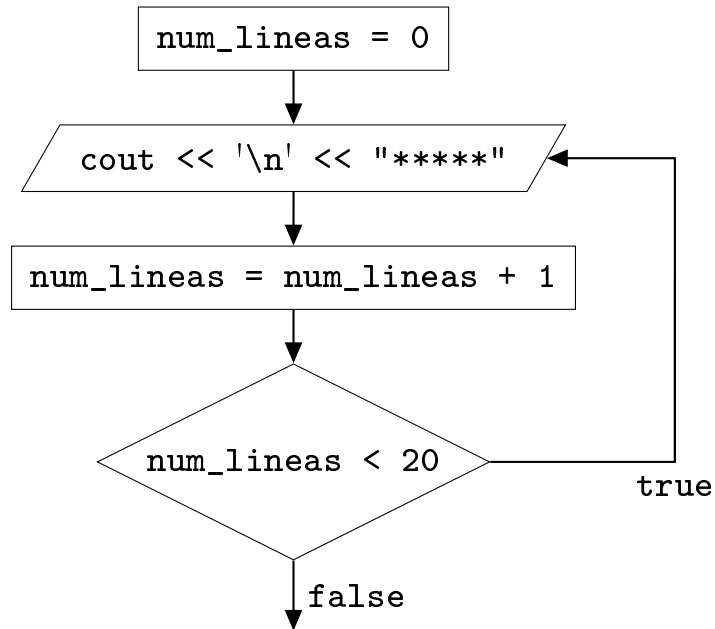
num_lineas = 1;

do{
    cout << '\n' << "*****" ;
    num_lineas = num_lineas + 1;    // nuevo = antiguo + 1
}while (num_lineas <= 20);
```

b) Post-test con condición $<$

```
num_lineas = 0;

do{
    cout << '\n' << "*****" ;
    num_lineas = num_lineas + 1;
}while (num_lineas < 20);
```



c) Pre-test con condición `<=`

```
num_lineas = 1;

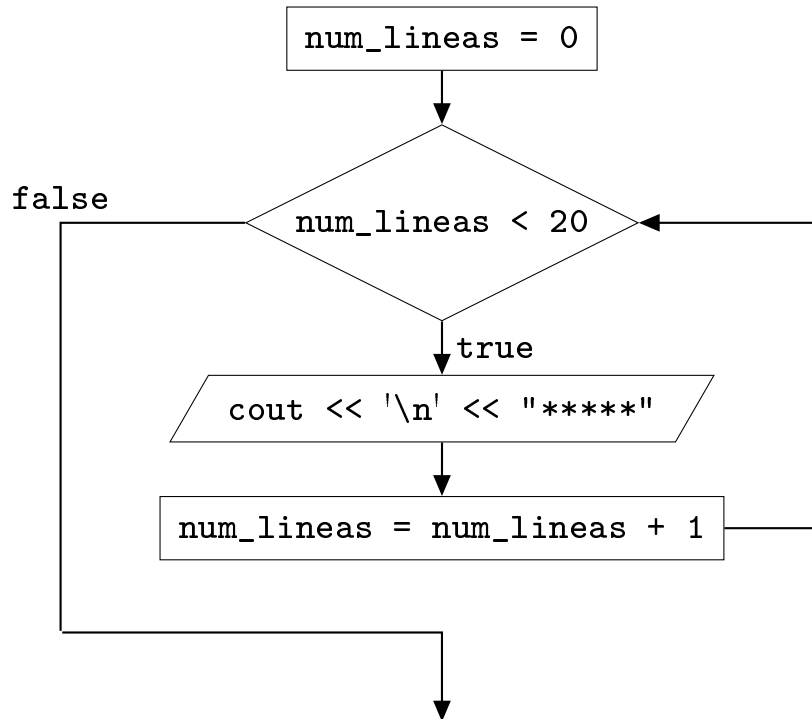
while (num_lineas <= 20){
    cout << '\n' << "*****" ;
    num_lineas = num_lineas + 1;
}
```

d) Pre-test con condición `<`

```
num_lineas = 0;    // Llevo 0 líneas impresas

while (num_lineas < 20){
    cout << '\n' << "*****" ;
    num_lineas = num_lineas + 1;
}
```





La opción d) es la preferible ya que es más fácil de entender: dentro del bucle, **justo antes** de comprobar la condición, la variable `total` contiene el número de líneas que han sido impresas. Además, una vez que termina el bucle la variable sigue valiendo lo mismo (ni uno más, ni uno menos). Finalmente, la inicialización a cero es coherente ya que al inicio aún no se ha impreso ninguna línea.

Nota:

Podríamos usar el operador de incremento:

```
while (num_lineas < 20){
    cout << '\\n' << "*****" ;
    num_lineas++;
}
```

¿Cuándo elegiremos un bucle pre-test y cuándo un post-test? Para ello, debemos responder a la siguiente pregunta: ¿Hay situaciones en las que no queremos que se ejecute el cuerpo del bucle?

Sí \Rightarrow pre-test

No \Rightarrow post-test

Ejemplo. Lea un número positivo `tope` e imprima `tope` líneas con 5 estrellas cada una.

```
cout << "\n¿Cuántas líneas de asteriscos quiere imprimir? ";

do{
    cin >> tope;
}while (tope < 0);    // Sale del filtro con un valor >= 0

num_lineas = 0;

do{
    cout << '\n' << "*****" ;
    num_lineas++;
}while (num_lineas < tope);
```

Problema: ¿Qué ocurre si `tope` vale 0?

Ejercicio. Resuelva el problema anterior con un bucle pre-test

Consejo: *Fomente el uso de los bucles pre-test. Lo usual es que haya algún caso en el que no queramos ejecutar el cuerpo del bucle ni siquiera una vez.*



Ejemplo. Sume los 100 primeros números positivos.

```
int suma, valor;  
  
valor = 1;  
  
while (valor <= 100){  
    suma = suma + valor;  
    valor++;  
}
```

En cada iteración, la instrucción

```
suma = suma + valor;
```

va acumulando en la variable `suma` las sumas anteriores.

Y la instrucción

```
valor++;
```

va sumándole 1 a la variable `valor`.

valor	suma	
1	1	= 1
2	1 + 2	= 3
3	3 + 3	= 6
4	6 + 4	= 10
...

Ejercicio. Calcule el número de dígitos que tiene un entero

57102 -> 5 dígitos

45 -> 2 dígitos

Algoritmo: Número de dígitos de un entero.

▷ **Entradas:** n

▷ **Salidas:** num_digitos

▷ **Descripción e implementación:**

Ir dividiendo n por 10 hasta llegar a una cifra

El número de dígitos será el número de iteraciones

http://decsai.ugr.es/jccubero/FP/II_numero_de_digitos.cpp

Ejemplo. Lea un entero `tope` positivo y escriba los pares \leq `tope`. ¿Cuál es el problema de esta solución?:

```
do{
    cin >> tope
}while (tope < 0);

par = 0;                // Primer candidato

while (par <= tope){
    par = par + 2;
    cout << par << " ";
}
```

Al final, escribe uno más. Cambiamos el orden de las instrucciones:

```
do{
    cin >> tope
}while (tope < 0);

par = 0;                // Primer candidato

while (par <= tope){    // ¿Es bueno?
    cout << par;        // Si => Imprímelo
    par = par + 2;      //      Calcular nuevo candidato
}                       // No => Salir
```

Si no queremos que salga 0, cambiaríamos la inicialización:

```
par = 2;                // Primer candidato
```

En el diseño de los bucles siempre hay que comprobar el correcto funcionamiento en los casos extremos (primera y última iteración)

II.2.1.3. Bucles para lectura de datos

En muchas ocasiones leeremos datos desde un dispositivo y tendremos que controlar una condición de parada. Habrá que controlar especialmente el primer y último valor leído.

Ejemplo. Realice un programa que sume una serie de valores leídos desde teclado, hasta que se lea el valor -1 (terminador = -1)

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

int main(){
    const int TERMINADOR = -1;
    int suma, numero;

    suma = 0;

    do{
        cin >> numero;
        suma = suma + numero;
    }while (numero != TERMINADOR);

    cout << "\nLa suma es " << suma;
}
```

Caso problemático: El último. Procesa el -1 y lo suma.

Una primera solución:

```
do{  
    cin >> numero;  
  
    if (numero != TERMINADOR)  
        suma = suma + numero;  
}while (numero != TERMINADOR);
```



Funciona, pero evalúa dos veces la misma condición, lo cual es ineficiente y, mucho peor, duplica código al repetir la expresión `numero != TERMINADOR`.

Soluciones:

- ▷ Usar variables lógicas.
- ▷ Técnica de *lectura anticipada* . Leemos el primer valor antes de entrar al bucle y comprobamos si hay que procesarlo (el primer valor podría ser ya el terminador)

Solución con una variable lógica:

```
bool seguir_leyendo;
```

```
do{
```

```
    cin >> numero;
```



```
// Leemos candidato
```

```
    seguir_leyendo = (numero != TERMINADOR);
```

```
    if (seguir_leyendo)
```

```
// Comprobamos si es el terminador
```

```
        suma = suma + numero;
```

```
// Lo procesamos
```

```
}while (seguir_leyendo);
```

La expresión que controla la condición de parada (numero != TERMINADOR); sólo aparece en un único sitio, por lo que si cambiase el criterio de parada, sólo habría que cambiar el código en dicho sitio.


Es verdad que repetimos la observación de la variable `seguir_leyendo` (en el `if` y en el `while`) pero no repetimos la evaluación de la expresión anterior.


Solución con lectura anticipada:

```
suma = 0;
cin >> numero;

while (numero != TERMINADOR) {
    suma = suma + numero;
    cin >> numero;
}

cout << "\nLa suma es " << suma;
```

 // Lectura anticipada del
// primer candidato

 // Comprobamos si es el terminador
// Lo procesamos
// Leemos siguiente candidato

http://decsai.ugr.es/jccubero/FP/II_suma_lectura_anticipada.cpp

A tener en cuenta:

- ▷ La primera vez que entra al bucle, la instrucción

```
while (numero != TERMINADOR)
```

hace las veces de un condicional. De esta forma, controlamos si hay que procesar o no el primer valor.

- ▷ Si el primer valor es el terminador, el algoritmo funciona correctamente.
- ▷ Hay cierto código repetido `cin >> numero;`, pero es aceptable. Mucho peor es repetir la expresión `numero != TERMINADOR` ya que es mucho más fácil que pueda cambiar en el futuro (porque cambie el criterio de parada)
- ▷ Dentro de la misma estructura repetitiva estamos mezclando las entradas (`cin >> numero;`) de datos con los cálculos (`suma = suma + numero;`), violando lo visto en la página 201. Por ahora no podemos evitarlo, ya que necesitaríamos almacenar los valores en un dato compuesto, para luego procesarlo. Lo resolveremos con el uso de vectores en el tema III.

Para no repetir código en los bucles que leen datos, normalmente usaremos la técnica de lectura anticipada:

```
cin >> dato;

while (dato no es último){
    procesar_dato
    cin >> dato;
}
```

A veces también puede ser útil introducir variables lógicas que controlen la condición de terminación de lectura:

```
do{
    cin >> dato;

    test_dato = ¿es el último?

    if (test_dato)
        procesar_dato
}while (test_dato);
```


Ejercicio. Lea enteros hasta llegar al cero. Imprima el número de pares e impares leídos.

Ejemplo. Retome el ejemplo de la subida salarial de la página 198. Creamos un programa para leer los datos de muchos empleados. El primer dato a leer será la experiencia. Si es igual a -1, el programa terminará.

```
const int TERMINADOR = -1;
.....
cin >> experiencia;

while (experiencia != TERMINADOR){
    cin >> salario_base; // Suponemos que el salario base es
                        // distinto en cada empleado

    cin >> edad;
    cin >> numero_hijos;
    .....
    if (es_aplicable_subida){
        .....
    }

    cin >> experiencia;
}
```

http://decsai.ugr.es/jccubero/FP/II_actualizacion_salarial_lectura_datos.cpp

Ejemplo. Lea datos enteros desde teclado hasta que se introduzca el 0. Calcule el número de valores introducidos y el mínimos de todos ellos. Vamos a ir mejorando la solución propuesta.

```
/*
Algoritmo:
    min contendrá el mínimo hasta ese momento

    Leer datos hasta llegar al terminador
    Actualizar el contador de valores introducidos
    Actualizar, en su caso, min
*/

cin >> dato;

while (dato != TERMINADOR){
    validos_introducidos++;

    if (dato < min)
        min = dato;

    cin >> dato;
}
```

¿Qué valor le damos a min la primera vez? ¿El mayor posible para garantizar que la expresión `dato < min` sea true la primera vez?

```
cin >> dato;
min = 32768;

while (dato != TERMINADOR){
    validos_introducidos++;

    if (dato < min)
```



```
    min = dato;  
  
    cin >> dato;  
}
```

¿Y si el compilador usa 32 bits en vez de 16 bits para representar un `int`?

¿Y si nos equivocamos al especificar el literal?

Recuerde lo visto en la página 35: evite siempre el uso de *números mágicos*.

Para resolver este problema, podríamos asignarle un valor dentro del bucle, detectando la primera iteración:

```
bool es_primera_vez;  
.....  
es_primera_vez = true;  
cin >> dato;  
  
while (dato != TERMINADOR){  
    validos_introducidos++;  
  
    if (es_primera_vez){  
        min = dato;  
        es_primera_vez = false;  
    }  
    else{  
        if (dato < min)  
            min = dato;  
    }  
  
    cin >> dato;  
}
```



Evite, en la medida de lo posible, preguntar por algo que sabemos de antemano que sólo va a ser verdadero en la primera iteración.

Realmente, la solución a nuestro problema es muy sencilla. Basta inicializar min al primer valor leído:

Algoritmo: Mínimo de varios valores.

- ▷ **Entradas:** Enteros hasta introducir un terminador
Salidas: El mínimo de ellos y el número de valores introducidos

- ▷ **Descripción e Implementación:**

```
/*  
Algoritmo:  
    Usamos una variable min, que contendrá el  
    mínimo hasta ese momento  
    Leer primer dato e inicializar min a dicho valor.  
    Leer datos hasta llegar al terminador  
        Actualizar el contador de valores introducidos  
        Actualizar, en su caso, min  
*/  
cin >> dato;  
min = dato;  
validos_introducidos = 0;  
  
while (dato != TERMINADOR){  
    validos_introducidos++;  
  
    if (dato < min)  
        min = dato;  
  
    cin >> dato;  
}
```

http://decsai.ugr.es/jccubero/FP/II_min_hasta_terminador.cpp

Nota:

Si se necesita saber el máximo valor entero representable, se puede recurrir a la función `numeric_limits<int>::max()`: de la biblioteca `limits`. En el ejemplo anterior, en vez de usar el número mágico `32768`, podríamos poner lo siguiente:

```
#include <limits>

.....

    cin >> dato;
    min = numeric_limits<int>::max();

    while (dato != TERMINADOR){
        validos_introducidos++;

        if (dato < min)
            min = dato;

        cin >> dato;
    }
```

Esta solución sí sería correcta. En cualquier caso, en el ejemplo que nos ocupa, la solución que hemos propuesto con la inicialización de `min` al primer dato leído es mucho más clara.

II.2.1.4. Bucles sin fin

Ejemplo. Imprima los divisores de un valor.

```
int divisor, valor, ultimo_divisor_posible;

cin >> valor;
ultimo_divisor_posible = valor / 2;
divisor = 2;

while (divisor <= ultimo_divisor_posible){
    if (valor % divisor == 0)
        cout << "\n" << divisor << " es un divisor de " << valor;

    divisor++;
}
```

¿Qué pasa si introducimos un else?

```
while (divisor <= ultimo_divisor_posible){
    if (valor % divisor == 0)
        cout << "\n" << divisor << " es un divisor de " << valor;
    else
        divisor++;
}
```

Es un bucle sin fin.

Debemos garantizar que en algún momento, la condición del bucle se hace falsa.

Tenga especial cuidado dentro del bucle con los condicionales que modifican alguna variable presente en la condición.

Ejemplo. ¿Cuántas iteraciones se producen?

```
contador = 1;

while (contador != 10) {
    contador = contador + 2;
}
```

Llega al máximo entero representable (2147483647). Al sumar 2, se produce un desbordamiento, obteniendo -2147483647. Al ser impar nunca llegará a 10, por lo que se producirá de nuevo la misma situación y el bucle no terminará.

Solución. Fomente el uso de condiciones de desigualdad:

```
contador = 1;

while (contador <= 10) {
    contador = contador + 2;
}
```


II.2.1.5. Condiciones compuestas

Es normal que necesitemos comprobar más de una condición en un bucle. Dependiendo del algoritmo necesitaremos conectarlas con `&&` o con `||`.

Ejemplo. Lea una opción de un menú. Sólo se admite s ó n.

```
char opcion;
do{
    cout << "¿Desea formatear el disco?";
    cin >> opcion;
}while ( opcion != 's'    opcion != 'S'
        opcion != 'n'    opcion != 'N' );
```

¿Cuándo quiero salir del bucle? Cuando *cualquiera* de las condiciones sea false. ¿Cual es el operador que cumple lo siguiente?:

```
false  Operador  <lo que sea>  =  false
true   Operador  true           =  true
```

Es el operador `&&`

Mejor aún si pasamos previamente el carácter a mayúscula y nos ahorramos dos condiciones:

```
do{
    cout << "Desea formatear el disco";
    cin >> opcion;
    opcion = toupper(opcion);
}while ( opcion != 'S' && opcion != 'N' );
```

Ejemplo. Calcule el máximo común divisor de dos números a y b.

Algoritmo: Máximo común divisor.

▷ **Entradas:** Los dos enteros a y b

Salidas: el entero `max_com_div`, máximo común divisor de a y b

▷ **Descripción e Implementación:**

```
/* Primer posible divisor = el menor de ambos
   Mientras divisor no divida a ambos,
   probar con el anterior */

if (b < a)
    menor = b;
else
    menor = a;

divisor = menor;

while (a % divisor != 0      b % divisor != 0)
    divisor --;

max_com_div = divisor;
```

¿Cuándo quiero salir del bucle? Cuando ambas condiciones, **simultáneamente**, sean false. En cualquier otro caso, entro de nuevo al bucle.

¿Cual es el operador que cumple lo siguiente?:

```
true   Operador <lo que sea> = true
false  Operador false  = false
```

Es el operador ||

```
while (a % divisor != 0 || b % divisor != 0)
    divisor --;

max_com_div = divisor;
```

En la construcción de bucles con condiciones compuestas, empiece planteando las condiciones simples que la forman. Piense cuándo queremos salir del bucle y conecte adecuadamente dichas condiciones simples, dependiendo de si nos queremos salir cuando todas simultáneamente sean false (conectamos con ||) o cuando cualquiera de ellas sea false (conectamos con &&)

Ejercicio. ¿Qué pasaría si a y b son primos relativos?

El uso de variables lógicas hará que las condiciones sean más fáciles de entender:

```
bool mcd_encontrado;  
.....  
mcd_encontrado = false;  
  
while (!mcd_encontrado){  
    if (a % divisor == 0 && b % divisor == 0)  
        mcd_encontrado = true;  
    else  
        divisor--;  
}  
  
max_com_div = divisor;
```

http://decsai.ugr.es/jccubero/FP/II_maximo_comun_divisor.cpp

Ambas soluciones son equivalentes. Formalmente:

```
a % divisor != 0 || b % divisor != 0  
equivale a  
!(a % divisor == 0 && b % divisor == 0)
```

Diseñe las condiciones compuestas de forma que sean fáciles de leer: en muchas situaciones, nos ayudará introducir variables lógicas a las que se les asignará un valor para salir del bucle cuando se verifique cierta condición de parada.

Ejercicio. ¿Qué pasaría si quitásemos el `else`?

II.2.1.6. Bucles que buscan

Una tarea típica en programación es buscar un valor. Si sólo estamos interesados en buscar uno, tendremos que salir del bucle en cuanto lo encontremos y así aumentar la eficiencia.

Ejemplo. Compruebe si un número entero positivo es primo. Para ello, debemos buscar un divisor suyo. Si lo encontramos, no es primo.

```
int valor, divisor, ultimo_divisor_posible;
bool es_primo;

cout << "Introduzca un numero natural: ";
cin >> valor;

es_primo = true;
divisor = 2;
ultimo_divisor_posible = divisor / 2;

while (divisor <= ultimo_divisor_posible){
    if (valor % divisor == 0)
        es_primo = false;
    divisor++;
}

if (es_primo)
    cout << valor << " es primo\n";
else{
    cout << valor << " no es primo\n";
    cout << "\nSu primer divisor es: " << divisor;
}
```



Funciona pero es ineficiente. Nos debemos salir del bucle en cuanto sepamos que no es primo. Usamos la variable `es_primo`.

```
int main(){
    int valor, divisor, ultimo_divisor_posible;
    bool es_primo;

    cout << "Introduzca un numero natural: ";
    cin >> valor;

    es_primo = true;
    divisor = 2;
    ultimo_divisor_posible = divisor / 2;

    while (divisor <= ultimo_divisor_posible && es_primo){
        if (valor % divisor == 0)
            es_primo = false;
        else
            divisor++;
    }

    if (es_primo)
        cout << valor << " es primo";
    else{
        cout << valor << " no es primo";
        cout << "\nSu primer divisor es: " << divisor;
    }
}
```



http://decsai.ugr.es/jccubero/FP/II_primo.cpp

Los algoritmos que realizan una búsqueda, deben salir de ésta en cuanto se haya encontrado el valor. Normalmente, usaremos una variable lógica para controlarlo.

IMPORTANT

Nota:

Al usar `else` garantizamos que al salir del bucle, la variable `divisor` contiene el primer divisor de `valor`

Ampliación:

Incluso podríamos quedarnos en `sqrt(valor)`, ya que si `valor` no es primo, tiene al menos un divisor menor que `sqrt(valor)`. En cualquier caso, `sqrt` es una operación costosa y habría que evaluar la posible ventaja en su uso.

```
es_primo = true;
ultimo_divisor_posible = sqrt(1.0 * valor);
                        // Necesario forzar casting a double
divisor = 2;

while (divisor <= ultimo_divisor_posible && es_primo){
    if (valor % divisor == 0)
        es_primo = false;
    else
        divisor++;
}
```



II.2.2. Programando como profesionales

II.2.2.1. Evaluación de expresiones dentro y fuera del bucle

En ocasiones, una vez terminado un bucle, es necesario comprobar cuál fue la condición que hizo que éste terminase. Veamos cómo hacerlo correctamente.

Ejemplo. Desde un sensor se toman datos de la frecuencia cardíaca de una persona. Emita una alarma cuando se encuentren fuera del rango [45, 120] indicando si es baja o alta. El sensor emite el valor -1 si la batería es insuficiente.

```
int main(){
    const int MIN_LATIDOS = 45;
    const int MAX_LATIDOS = 120;
    const int SIN_BATERIA = -1;
    int latidos;

    // La siguiente versión repite código:

    do{
        cin >> latidos;
    }while (MIN_LATIDOS <= latidos && latidos <= MAX_LATIDOS &&
           latidos != SIN_BATERIA);

    if (latidos == SIN_BATERIA)
        cout << "\nBatería baja";
    else if (latidos < MIN_LATIDOS || latidos > MAX_LATIDOS)
        cout << "\nNúmero de latidos anormal: " << latidos;
    else
        cout << "\nError desconocido";
}
```



Se repite código ya que, por ejemplo, la expresión `latidos < MIN_LATIDOS` equivale a `!(MIN_LATIDOS <= latidos)` Para resolverlo, introducimos variables lógicas intermedias:

```
int main(){
    .....
    bool frecuencia_cardiaca_anormal, sensor_sin_bateria;

    do{
        cin >> latidos;
        frecuencia_cardiaca_anormal = latidos < MIN_LATIDOS
                                   ||
                                   latidos > MAX_LATIDOS;
        sensor_sin_bateria = latidos == SIN_BATERIA;
    }while (!frecuencia_cardiaca_anormal && !sensor_sin_bateria);

    if (sensor_sin_bateria)
        cout << "\nBatería baja";
    else if (frecuencia_cardiaca_anormal)
        cout << "\nNúmero de latidos anormal: " << latidos;
    else
        cout << "\nError desconocido";
}
```



Y mejor aún, podríamos usar un enumerado para distinguir todas las situaciones posibles (frecuencia normal, anormalmente baja y anormalmente alta):

http://decsai.ugr.es/jccubero/FP/II_frecuencia_cardiaca.cpp

Si una vez que termina un bucle controlado por varias condiciones, necesitamos saber cuál de ellas hizo que terminase éste, introduciremos variables intermedias para determinarlo. Nunca repetiremos la evaluación de condiciones.

II.2.2.2. Bucles que no terminan todas sus tareas

Ejemplo. Queremos leer las notas de un alumno y calcular la media aritmética. El alumno tendrá un máximo de cuatro calificaciones. Si tiene menos de cuatro, introduciremos cualquier negativo para indicarlo.

```
suma = 0;
cin >> nota;
total_introducidos = 1;

while (nota >= 0 && total_introducidos <= 4){
    suma = suma + nota;
    cin >> nota;
    total_introducidos++;
}

media = suma/total_introducidos;
```



Problema: Lee la quinta nota y se sale, pero ha tenido que leer dicho valor.

Solución: O bien cambiamos la inicialización de `total_introducidos` a 0, o bien leemos hasta menor estricto de 4; pero entonces, el cuarto valor (en general el último) hay que procesarlo fuera del bucle.

```
suma = 0;
cin >> nota;
total_introducidos = 1;

while (nota >= 0 && total_introducidos < 4){
    suma = suma + nota;
    cin >> nota;
    total_introducidos++;
}

suma = suma + nota;
media = suma/total_introducidos;
```



Problema: Si el valor es negativo lo suma. Lo resolvemos con un condicional:

```
suma = 0;
cin >> nota;
total_introducidos = 1;

while (nota >= 0 && total_introducidos < 4){
    suma = suma + nota;
    cin >> nota;
    total_introducidos++;
}

if (nota > 0)
    suma = suma + nota;

media = suma/total_introducidos;
```



Además, no debemos aumentar total_introducidos si es un negativo:

```
if (nota > 0)
    suma = suma + nota;
else
    total_introducidos--;

media = suma/total_introducidos;
```



Podemos observar la complejidad (por no decir chapucería) innecesaria que ha alcanzado el programa.

Replanteamos desde el inicio la solución y usamos variables lógicas:



```
int main(){
    const int TOPE_NOTAS = 4;
    int nota, suma, total_introducidos;
    double media;
    bool tope_alcanzado, es_correcto;

    cout << "Introduzca un máximo de " << TOPE_NOTAS
         << " notas, o cualquier negativo para finalizar.\n ";

    suma = 0;
    total_introducidos = 0;
    es_correcto = true;
    tope_alcanzado = false;

    do{
        cin >> nota;

        if (nota < 0)
            es_correcto = false;
        else{
            suma = suma + nota;
            total_introducidos++;

            if (total_introducidos == TOPE_NOTAS)
                tope_alcanzado = true;
        }
    }while (es_correcto && !tope_alcanzado);

    media = suma/(1.0 * total_introducidos);    // Si total_introducidos es 0
                                                // media = infinito

    if (total_introducidos == 0)
        cout << "\nNo se introdujo ninguna nota";
    else
```

```
    cout << "\nMedia aritmética = " << media;  
}
```

http://decsai.ugr.es/jccubero/FP/II_notas.cpp

Construya los bucles de forma que no haya que arreglar nada después de su finalización

II.2.2.3. Estilo de codificación

La siguiente implementación del anterior algoritmo es nefasta ya que cuesta mucho trabajo entenderla debido a los identificadores elegidos, a las tabulaciones mal hechas y a la falta de líneas en blanco que separen visualmente bloques de código.

```
int main(){
    const int T = 4;
    int v, aux, contador;
    double resultado;
    bool seguir_1, seguir_2;
    aux = 0;
    contador = 0;
    seguir_1 = true;
    seguir_2 = false;
    do{cin >> v;
    if (v < 0)
    seguir_1 = false;
    else{
        aux = aux + v;
        contador++;
        if (contador == T)
            seguir_2 = true;
    }
}while (seguir_1 && !seguir_2);
resultado = aux/(1.0*contador);
if (contador == 0)
    cout << "\nNo se introdujeron valores";
else
    cout << "\nMedia aritmética = " << resultado;
}
```



II.2.3. Bucles controlador por contador

II.2.3.1. Motivación

Se utilizan para repetir un conjunto de sentencias un número de veces fijado de antemano. Se necesita una variable contadora, un valor inicial, un valor final y un incremento.

Ejemplo. Calcule la media aritmética de cinco enteros leídos desde teclado.

```
int main(){
    int contador, valor, suma, inicio, final;
    double media;

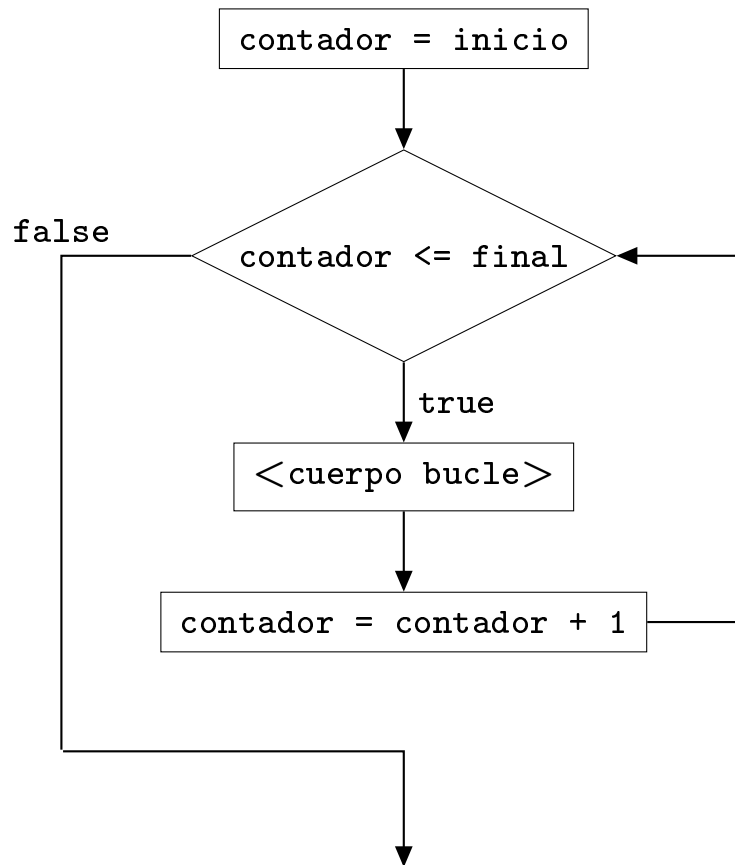
    inicio = 1;
    final = 5;
    suma = 0;
    contador = inicio;

    while (contador <= final){
        cout << "\nIntroduce un número ";
        cin >> valor;
        suma = suma + valor;

        contador = contador + 1;
    }

    media = suma / (final * 1.0);
    cout << "\nLa media es " << media;
}
```


El diagrama de flujo correspondiente es:

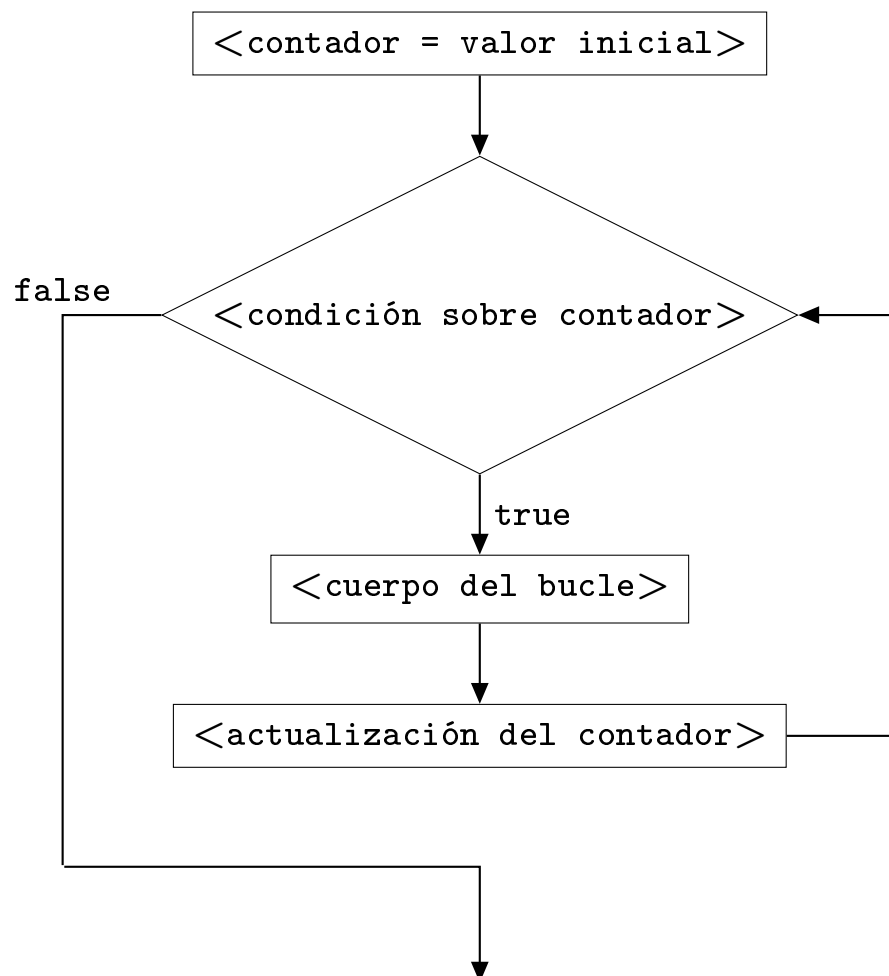


Vamos a implementar el mismo diagrama, pero con otra sintaxis (usando el bucle `for`).

II.2.3.2. Formato

La sentencia `for` permite la construcción de una forma compacta de los ciclos controlados por contador, aumentando la legibilidad del código.

```
for (<contador = valor inicial> ; <condición sobre contador>  
    ; <actualización del contador> )  
    <cuerpo del bucle>
```



Usaremos los bucles `for` cuando sepamos, antes de entrar al bucle, el número de iteraciones que se tienen que ejecutar.

Ejemplo. Calcule la media aritmética de cinco enteros leídos desde teclado.

```
int main(){
    int contador, valor, suma, inicio, final;
    double media;

    inicio = 1;
    final = 5;
    suma = 0;

    for (contador = inicio ; contador <= final ; contador = contador + 1){
        cout << "\nIntroduce un número ";
        cin >> valor;
        suma = suma + valor;
    }

    media = suma / (final*1.0);
    cout << "\nLa media es " << media;
}
```

Como siempre, si sólo hay una sentencia dentro del bucle, no son necesarias las llaves.

```
for (contador = inicio ; contador <= final ; contador = contador + 1){  
    cout << "\nIntroduce un número ";  
    cin >> valor;  
    suma = suma + valor;  
}
```

- ▷ **La primera parte, `contador = inicio`, es la asignación inicial de la variable contadora. Sólo se ejecuta una única vez (cuando entra al bucle por primera vez)**
- ▷ **La segunda parte, `contador <= final`, es la condición de continuación del bucle.**
- ▷ **La tercera parte, `contador = contador + 1`, es la sentencia de actualización de la variable contadora.**

A tener en cuenta:

- ▷ **`contador = contador + 1` aumenta en 1 el valor de `contador` en cada iteración. Por abreviar, suele usarse `contador++` en vez de `contador = contador + 1`**

```
for (contador = inicio ; contador <= final ; contador++)
```

Podemos usar cualquier otro incremento:

```
contador = contador + 4;
```

- ▷ **Si usamos como condición**

```
contador < final
```

habrá menos iteraciones. Si el incremento es 1, se producirá una iteración menos.

- ▷ **También pueden usarse incrementos negativos. En este caso, la condición de terminación del bucle tendrá que ser del tipo `contador >= final` o `contador > final`**

Ejercicio. Queremos imprimir los números del 100 al 1. Encuentre errores en este código:

```
For {x = 100, x>=1, x++}  
    cout << x << " ";
```

Ejemplo. Imprima los pares que hay en el intervalo $[-10, 10]$

```
int candidato;  
  
num_pares = 0;  
  
for (candidato = -10; candidato <= 10; candidato++) {  
    if (candidato % 2 == 0)  
        cout << candidato << " ";  
}
```

Ejercicio. Resuelva el anterior problema con otro bucle distinto.

Ejemplo. Imprima una línea con 10 asteriscos.

```
int i;

for (i = 1; i <= 10; i++)
    cout << "*";
```

¿Con qué valor sale la variable `i`? 11

Cuando termina un bucle `for`, la variable contadora se queda con el primer valor que hace que la condición del bucle sea falsa.

¿Cuántas iteraciones se producen en un `for`?

▷ Si incremento = 1, $\text{inicio} \leq \text{final}$ y `contador <= final`

`final - inicio + 1`

▷ Si incremento = 1, $\text{inicio} \leq \text{final}$ y `contador < final`

`final - inicio`

```
for (i = 0; i < 10; i++)    --> 10 - 0 = 10
    cout << "*";
```

```
for (i = 12; i > 2; i--)    --> 12 - 2 = 10
    cout << "*";
```

```
for (i = 10; i >= 1; i--)   --> 10 - 1 + 1 = 10
    cout << "*";
```

Ampliación:



Número de iteraciones con incrementos cualesquiera.

Si es del tipo `contador <= final`, tenemos que contar cuántos intervalos de longitud igual a `incremento` hay entre los valores `inicio` y `final`. El número de iteraciones será uno más.

En el caso de que `contador < final`, habrá que contar el número de intervalos entre `inicio` y `final - 1`.

El número de intervalos se calcula a través de la división entera.

En resumen, considerando incrementos positivos:

- ▷ Si el bucle es del tipo `contador <= final` el número de iteraciones es $(\text{final} - \text{inicio}) / \text{incremento} + 1$ siempre y cuando sea `inicio <= final`. En otro caso, hay 0 iteraciones.
- ▷ Si el bucle es del tipo `contador < final` el número de iteraciones es $(\text{final} - 1 - \text{inicio}) / \text{incremento} + 1$ siempre y cuando sea `inicio < final`. En otro caso, hay 0 iteraciones.
- ▷ De forma análoga se realizan los cálculos con incrementos negativos (en cuyo caso, el valor inicial ha de ser mayor o igual que el final).

Nota:

Cuando las variables usadas en los bucles no tienen un significado especial podremos usar nombres cortos como `i`, `j`, `k`

Ejercicio. ¿Qué salida producen los siguientes trozos de código?

```
int i, suma_total;
suma_total = 0;

for (i = 1 ; i <= 10; i++)
    suma_total = suma_total + 3;
```

```
suma_total = 0;

for (i = 5 ; i <= 36 ; i++)
    suma_total++;
```

```
suma_total = 0;

for (i = 5 ; i <= 36 ; i = i+1)
    suma_total++;
```

```
suma_total = 0;
i = 5;

while (i <= 36){
    suma_total++;
    i = i+1;
}
```

II.2.4. Ámbito de un dato (revisión)

Ya vimos en la página 175 que un dato puede declararse en un bloque condicional y su *ámbito (scope)* es dicho bloque. En general, un dato se puede declarar en cualquier bloque delimitado por { y }. En particular, también en un bloque de una estructura repetitiva:

- ▷ En un bucle controlado por condición, el dato declarado en el bloque no se conoce fuera de él; en particular, no se puede usar en la condición.

```
while (i < 20){           // Error de compilación (i fuera de ámbito)
    int i = 0;
    cout << i << " ";
    i++;
}
```

Además, el dato pierde el valor antiguo en cada iteración:

```
int impresos = 0;

while (impresos < 20){    😞
    int i = 0;
    cout << i << " ";    // Imprime 0 0 0 ...
    impresos++;
    i++;
}
```

Cada vez que entra en el bucle se asigna `i = 0`.

- ▷ El comportamiento es distinto en un bucle `for`. El bucle conserva el valor de la iteración anterior:

```
cout << i;                // Error de compilación (i fuera de ámbito)

for (int i = 0; i < 20; i++)    😊
    cout << i << " "; // Imprime 0 1 2 ...
```

Sólo se asigna `i = 0` en la primera iteración. El uso de este tipo de variables será bastante usual.

Debemos tener cuidado con la declaración de variables con igual nombre que otra definida en un ámbito *superior*.

Prevalece la de ámbito más restringido → *prevalencia de nombre (name hiding)*

```
int main(){
    int suma = 0;                // <- suma (ámbito: main)
    int ultimo;
    int a_sumar = 0;

    cin >> ultimo;

    while (a_sumar < ultimo){
        int suma = 0;            // <- suma (ámbito: bloque while)
        suma = suma + a_sumar;    // <- suma (ámbito: bloque while)
        a_sumar++;
    }

    cout << suma;                // <- suma (ámbito: main)
                                // Imprime 0    😞
}
```

En resumen:

Cuando necesitemos puntualmente variables intermedias para realizar nuestros cálculos, será útil definirlas en el ámbito del bloque de instrucciones correspondiente.

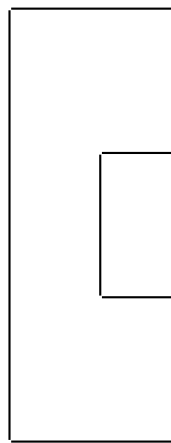
Pero hay que tener cuidado si es un bloque `while` ya que pierde el valor en cada iteración.

Esto no ocurre con las variables contadoras del bucle `for`, que recuerdan el valor que tomó en la iteración anterior.

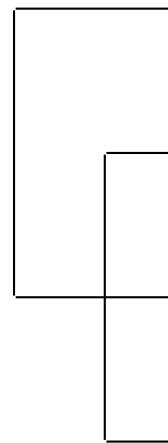
II.2.5. Anidamiento de bucles

Dos bucles se encuentran anidados, cuando uno de ellos está en el bloque de sentencias del otro.

En principio no existe límite de anidamiento, y la única restricción que se debe satisfacer es que deben estar completamente inscritos unos dentro de otros.



(a) Anidamiento correcto



(b) Anidamiento incorrecto

En cualquier caso, un factor determinante a la hora de determinar la rapidez de un algoritmo es la profundidad del anidamiento. Cada vez que anidamos un bucle dentro de otro, la ineficiencia se dispara.

Ejemplo. Imprima la tabla de multiplicar de los `TOPE` primeros números.

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

int main(){
    const int TOPE_IZDA = 3;
    const int TOPE_DCHA = 3;
    int izda, dcha;

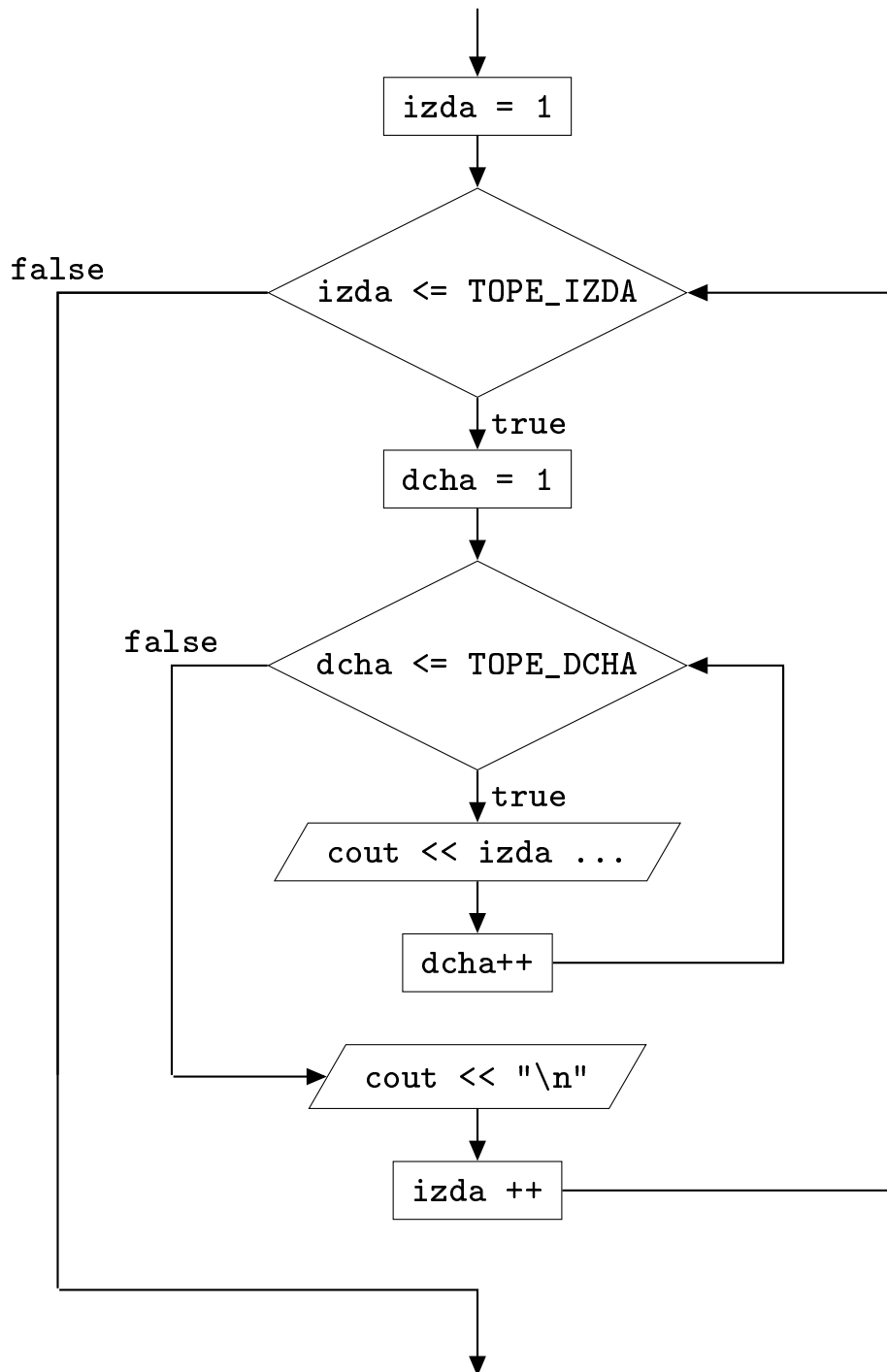
    cout << "Impresión de la tabla de multiplicar "
         << TOPE_IZDA << " x " << TOPE_DCHA << "\n";

    for (izda = 1 ; izda <= TOPE_IZDA ; izda++) {
        for (dcha = 1 ; dcha <= TOPE_DCHA ; dcha++)
            cout << izda << "*" << dcha << "=" << izda * dcha << "  ";
        cout << "\n";
    }
}
```

http://decsai.ugr.es/jccubero/FP/II_tabla_de_multiplicar.cpp

	dcha=1	dcha=2	dcha=3
izda=1	1*1 = 1	1*2 = 2	1*3 = 3
izda=2	2*1 = 2	2*2 = 4	2*3 = 6
izda=3	3*1 = 3	3*2 = 6	3*3 = 9

Observe que cada vez que avanza `izda` y entra de nuevo al bucle, la variable `dcha` vuelve a inicializarse a 1



Al diseñar bucles anidados, hay que analizar cuidadosamente las variables que hay que reiniciar antes de entrar a los bucles más internos

Ejemplo. ¿Qué salida produce el siguiente código?

```
iteraciones = 0;
suma = 0;

for (i = 1 ; i <= n; i++){
    for (j = 1 ; j <= n; j++){
        suma = suma + j;
        iteraciones++;
    }
}
```

Número de iteraciones: n^2

Valor de la variable suma. Supongamos $n = 5$. Valores que va tomando j :

```
1 + 2 + 3 + 4 + 5 +
1 + 2 + 3 + 4 + 5 +
1 + 2 + 3 + 4 + 5 +
1 + 2 + 3 + 4 + 5 +
1 + 2 + 3 + 4 + 5
```

$\text{suma} = 5 * (1 + 2 + 3 + 4 + 5)$. **En general:**

$$\text{suma} = n \sum_{i=1}^{i=n} i = n \frac{n^2 + n}{2} = \frac{n^3 + n^2}{2}$$

Si n es 5, suma se quedará con 75

Ejemplo. ¿Qué salida produce el siguiente código?

```
iteraciones = 0;
suma = 0;

for (i = 1 ; i <= n; i++){
    for (j = i ; j <= n; j++){
        suma = suma + j;
        iteraciones++;
    }
}
```

Número de iteraciones:

$$n + (n - 1) + (n - 2) + \dots + 1 = \sum_{i=1}^{i=n} i = \frac{n^2 + n}{2} < n^2$$

Valor de la variable suma. Supongamos $n = 5$. Valores que va tomando j :

```
1 + 2 + 3 + 4 + 5 +
  2 + 3 + 4 + 5 +
    3 + 4 + 5 +
      4 + 5 +
        5
```

$$\text{suma} = 5 * 5 + 4 * 4 + 3 * 3 + 2 * 2 + 1 * 1 = \sum_{i=1}^{i=n} i^2 = \frac{1}{6}n(n + 1)(2n + 1)$$

Si n es 5, suma se quedará con 55

Ejercicio. Escriba un programa que lea cuatro valores de tipo `char` (`min_izda`, `max_izda`, `min_dcha`, `max_dcha`) e imprima las parejas que pueden formarse con un elemento del conjunto `{min_izda ... max_izda}` y otro elemento del conjunto `{min_dcha ... max_dcha}`. Por ejemplo, si

```
min_izda = b
max_izda = d
```

```
min_dcha = j
max_dcha = m
```

el programa debe imprimir las parejas que pueden formarse con un elemento de `{b c d}` y otro elemento de `{j k l m}`, es decir:

```
bj bk bl bm
cj ck cl cm
dj dk dl dm
```

http://decsai.ugr.es/jccubero/FP/II_Parejas.cpp

Ejemplo. Extienda el ejercicio que comprobaba si un número es primo (página 243) e imprima en pantalla los primos menores que un entero.

```
int main(){
    int entero, posible_primo, divisor, ultimo_divisor_posible;
    bool es_primo;

    cout << "Introduzca un entero ";
    cin >> entero;
    cout << "\nLos primos menores que " << entero << " son:\n";

    /*
    Recorremos todos los números menores que entero
    Comprobamos si dicho número es primo
    */

    for (posible_primo = entero - 1 ; posible_primo > 1 ; posible_primo--){
        es_primo = true;
        divisor = 2;
        ultimo_divisor_posible = posible_primo / 2;

        while (divisor <= ultimo_divisor_posible && es_primo){
            if (posible_primo % divisor == 0)
                es_primo = false;
            else
                divisor++;
        }

        if (es_primo)
            cout << posible_primo << " ";
    }
}
```

http://decsai.ugr.es/jccubero/FP/II_imprimir_primos.cpp

Ejemplo. El *Teorema fundamental de la Aritmética* (Euclides 300 A.C/Gauss 1800) nos dice que podemos expresar cualquier entero como producto de factores primos.

Imprima en pantalla dicha descomposición.

n	primo
360	2
180	2
90	2
45	3
15	3
5	5
1	

Fijamos un valor de primo cualquiera. Por ejemplo `primo = 2`

```
// Dividir n por primo cuantas veces sea posible  
// n es una copia del original
```

```
primo = 2;
```

```
while (n % primo == 0){  
    cout << primo << " ";  
    n = n / primo;  
}
```

Ahora debemos pasar al siguiente primo `primo` **y volver a ejecutar el bloque anterior. Condición de parada:** `n >= primo` **o bien** `n > 1`

```
Mientras n > 1
```

```
    Dividir n por primo cuantas veces sea posible  
    primo = siguiente primo mayor que primo
```

¿Cómo pasamos al siguiente primo?

```
Repite mientras !es_primo
    primo++;
    es_primo = Comprobar si primo es un número primo
```

La comprobación de ser primo o no la haríamos con el algoritmo que vimos en la página 243. Pero no es necesario. Hagamos simplemente `primo++`:

```
/*
Mientras n > 1
    Dividir n por primo cuantas veces sea posible
    primo++
*/

primo = 2;

while (n > 1){
    while (n % primo == 0){
        cout << primo << " ";
        n = n / primo;
    }
    primo++;
}
```

¿Corremos el peligro de intentar dividir `n` por un valor `primo` que no sea primo? No. Por ejemplo, `n=40`. Cuando `primo` sea 4, ¿podrá ser `n` divisible por 4, es decir `n%4==0`? Después de dividir todas las veces posibles por 2, me queda `n=5` que ya no es divisible por 2, ni por tanto, por ningún múltiplo de 2. En general, al evaluar `n%primo`, `n` ya ha sido dividido por todos los múltiplos de `primo`.

Nota. Podemos sustituir `primo++` por `primo = primo+2` (tratando el primer caso `primo = 2` de forma aislada)

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

int main(){
    int entero, n, primo;

    cout << "Descomposición en factores primos";
    cout << "\nIntroduzca un entero ";
    cin >> entero;

    /*
    Copiar entero en n

    Mientras n > 1
        Dividir n por primo cuantas veces sea posible
        primo++
    */

    n = entero;
    primo = 2;

    while (n > 1){
        while (n % primo == 0){
            cout << primo << " ";
            n = n / primo;
        }
        primo++;
    }
}
```

http://decsai.ugr.es/jccubero/FP/II_descomposicion_en_primos.cpp

II.3. Particularidades de C++

C++ es un lenguaje muy versátil. A veces, demasiado . . .

II.3.1. Expresiones y sentencias son similares

II.3.1.1. El tipo `bool` como un tipo entero

En C++, el tipo lógico es compatible con un tipo entero. Cualquier expresión entera que devuelva el cero, se interpretará como `false`. Si devuelve cualquier valor distinto de cero, se interpretará como `true`.

```
bool var_logica;

var_logica = false;
var_logica = (4 > 5); // Correcto: resultado false
var_logica = 0;      // Correcto: resultado 0 (false)

var_logica = (4 < 5); // Correcto: resultado true
var_logica = true;
var_logica = 2;      // Correcto: resultado 2 (true)
```

Nota. Normalmente, al ejecutar `cout << false`, se imprime en pantalla un cero, mientras que `cout << true` imprime un uno.

La dualidad entre los tipos enteros y lógicos nos puede dar quebraderos de cabeza en los condicionales

```
int dato = 4;
if (! dato < 5)
    cout << dato << " es mayor o igual que 5";
else
    cout << dato << " es menor de 5";
```



El operador ! tiene más precedencia que <. Por lo tanto, la evaluación es como sigue:

$! \text{ dato} < 5 \Leftrightarrow (!\text{dato}) < 5 \Leftrightarrow (4 \text{ equivale a true}) (!\text{true}) < 5 \Leftrightarrow$
 $\Leftrightarrow \text{false} < 5 \Leftrightarrow 0 < 5 \Leftrightarrow \text{true}$

¡Imprime 4 es mayor o igual que 5!

Para resolver este problema basta usar paréntesis:

```
if (! (dato < 5))
    cout << dato << " es mayor o igual que 5";
else
    cout << dato << " es menor de 5";
```

o mejor, simplificando la expresión siguiendo el consejo de la página 149

```
if (dato >= 5)
    cout << dato << " es mayor o igual que 5";
else
    cout << dato << " es menor de 5";
```

Ejercicio. ¿Qué problema hay en este código? ¿Cómo lo resolvería?

```
bool es_menor;

es_menor = 2 <= 3 <= 4;
```

II.3.1.2. El operador de asignación en expresiones

El operador de asignación = se usa en sentencias del tipo:

```
valor = 7;
```

Pero además devuelve un valor: el resultado de la asignación. Así pues, `valor = 7` **es una expresión** que devuelve 7

```
un_valor = otro_valor = valor = 7;
```

Esto producirá fuertes dolores de cabeza cuando por error usemos una expresión de asignación en un condicional:

```
valor = 5;

if (valor = 7)
    <acciones if>    // Siempre se ejecuta este bloque!
else
    <acciones else>

// Además, valor se queda con 7
```



`valor = 7` devuelve 7. Al ser distinto de cero, es `true`. Por tanto, se ejecuta el bloque `if` (y además `valor` se ha modificado con 7)

Otro ejemplo:

```
a = 7;

if (a = 0)
    cout << "\nRaíz= " << -c/b;           // Nunca se ejecuta!
else{
    r1 = -b + sqrt(b*b - 4*a*c) / (2*a) ; // Error lógico
```


II.3.1.3. El operador de igualdad en sentencias

C++ permite que una expresión constituya una sentencia 😞

Esta particularidad no aporta ningún beneficio salvo en casos muy específicos y sin embargo nos puede dar quebraderos de cabeza. Así pues, el siguiente código compila perfectamente:

```
int entero;  
4 + 3;
```

C++ evalúa la expresión **entera** `4 + 3;`, devuelve 7 y no hace nada con él, prosiguiendo la ejecución del programa.

Otro ejemplo:

```
int entero;  
entero == 7;
```

C++ evalúa la expresión **lógica** `entero == 7;`, devuelve `true` y no hace nada con él, prosiguiendo la ejecución del programa.

II.3.1.4. El operador de incremento en expresiones

El operador ++ (y --) puede usarse dentro de una expresión.

Si se usa en forma postfija, primero se evalúa la expresión y luego se incrementa la variable.

Si se usa en forma prefija, primero se incrementa la variable y luego se evalúa la expresión.

Ejemplo. El siguiente condicional expresa una condición del tipo *Comprueba si el siguiente es igual a 10*

```
variable = 9;
if (variable + 1 == 10)
    cout << variable;           // Imprime 9
cout << " " << variable;       // Imprime 9
```

El siguiente condicional expresa una condición del tipo *Comprueba si el actual es igual a 10 y luego increméntalo*

```
variable = 9;
if (variable++ == 10)
    cout << variable;           // No entra
cout << " " << variable;       // Imprime 10
```

El siguiente condicional expresa una condición del tipo *Incrementa el actual y comprueba si es igual a 10*

```
variable = 9;
if (++variable == 10)
    cout << variable;           // Imprime 10
cout << " " << variable;       // Imprime 10
```

Consejo: *Evite el uso de los operadores ++ y -- en la expresión lógica de una sentencia condicional, debido a las sutiles diferencias que hay en su comportamiento, dependiendo de si se usan en formato prefijo o postfijo*



II.3.2. El bucle for en C++

II.3.2.1. Bucles for con cuerpo vacío

El siguiente código no imprime los enteros del 1 al 20. ¿Por qué?

```
for (x = 1; x <= 20; x++);  
    cout << x;
```

Realmente, el código bien tabulado es:

```
for (x = 1; x <= 20; x++)  
    ;  
    cout << x;
```

Evite este tipo de bucles con sentencias vacías ya que son muy oscuros.

II.3.2.2. Bucles for con sentencias de incremento incorrectas

El siguiente código produce un error lógico:

```
for (par = -10; par <= 10; par + 2) // en vez de par = par + 2  
    num_pares++;
```

equivale a:

```
par = -10;  
while (par <= 10){  
    num_pares++;  
    par + 2;  
}
```

Compila correctamente pero la sentencia `par + 2;` no incrementa `par` (recuerde lo visto en la página 278) Resultado: bucle infinito.

II.3.2.3. Modificación del contador

Únicamente mirando la cabecera de un bucle `for` sabemos cuántas iteraciones se van a producir (recuerde lo visto en la página 260). Por eso, en los casos en los que sepamos de antemano cuántas iteraciones necesitamos, usaremos un bucle `for`. En otro caso, usaremos un bucle `while` ó `do while`.

En el caso de que modifiquemos dentro del bucle `for` el valor de la variable controladora o el valor final de la condición, ya no sabremos de antemano cuántas iteraciones se van a ejecutar. Sin embargo, C++ no impone dicha restricción. Será responsabilidad del programador.

Ejercicio. Sume los divisores de `valor`. ¿Dónde está el fallo en el siguiente código? Proponga una solución.

```
suma = 0;
tope = valor/2;

for (divisor = 2; divisor <= tope ; divisor++) {
    if (valor % divisor == 0)
        suma = suma + divisor;

    divisor++;
}
```

Ejemplo. Lea seis notas y calcule la media aritmética. Si se introduce cualquier valor que no esté en el rango $[0, 10]$ se interrumpirá la lectura.

```
double nota, media = 0.0;
int total_introducidos = 0;

for (i = 0; i < 6; i++){
    cin >> nota;

    if (0 <= nota && nota <= 10){
        total_introducidos++;
        media = media + nota;
    }
    else
        i = 7;
}

media = media / total_introducidos;
```



El código produce un resultado correcto de cara al usuario pero es muy oscuro para otro programador. Hay que escudriñar en el cuerpo del bucle para ver todas las condiciones involucradas en la ejecución de éste. Esta información debería estar en la cabecera del `for`.

La solución pasa por usar el bucle `while`:

```
double nota, media = 0.0;
int total_introducidos = 0;
int i;

cin >> nota;
i = 0;

while (i < 6 && 0 <= nota && nota <= 10){
    total_introducidos++;
    media = media + nota;
    i++;
    cin >> nota;
}

media = media / total_introducidos;
```



En resumen:

**No se debe modificar el valor de la variable controladora,
ni el valor final dentro del cuerpo del bucle `for`.**

**En particular, nunca nos saldremos de un bucle `for`
asignándole un valor extremo a la variable contadora.**

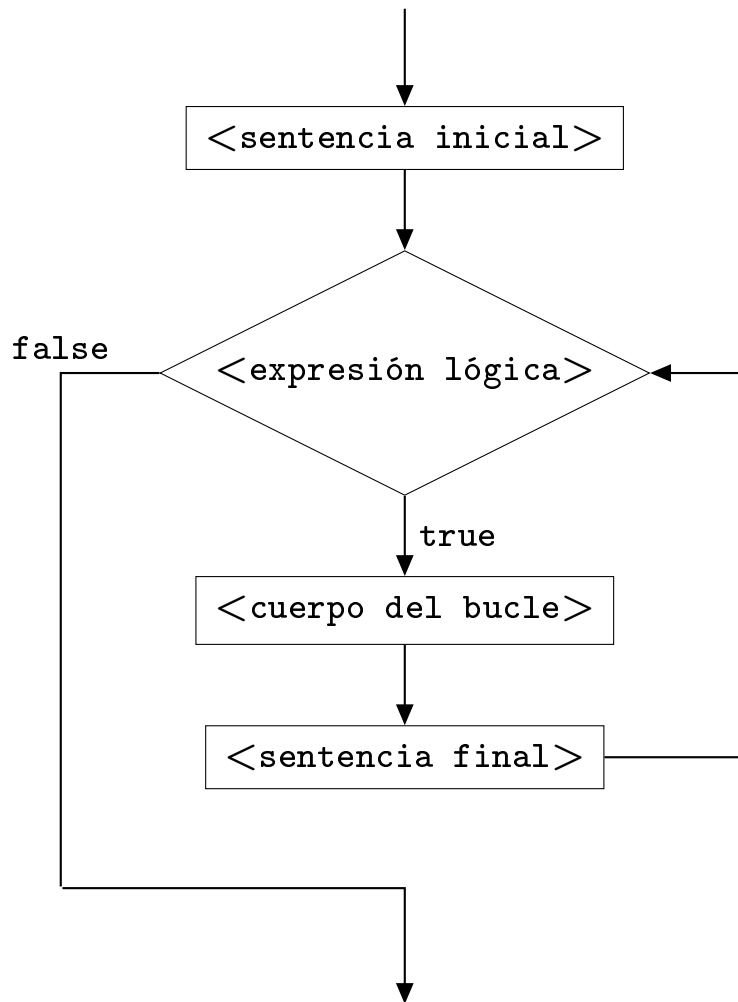


II.3.2.4. El bucle `for` como ciclo controlado por condición

Si bien en muchos lenguajes tales como PASCAL, FORTRAN o BASIC el comportamiento del ciclo `for` es un bucle controlado por contador, en C++ es un ciclo más versátil que permite cualquier tipo de expresiones, involucren o no a un contador:

```
for (<sentencia inicial> ; <expresión lógica>  
    ; <sentencia final> )  
    <cuerpo del bucle>
```

- ▷ **<sentencia inicial>** es la sentencia que se ejecuta antes de entrar al bucle,
- ▷ **<expresión lógica>** es cualquier condición que verifica si el ciclo debe terminar o no,
- ▷ **<sentencia final>** es la sentencia que se ejecuta antes de volver a la expresión lógica para comprobar su valor.



Por tanto, la condición impuesta en el ciclo no tiene por qué ser de la forma `contador < final`, sino que puede ser cualquier tipo de condición. Veamos en qué situaciones es útil esta flexibilidad.

Ejemplo. Retomamos el ejemplo de la media de las notas. Podemos recuperar la versión con un bucle `for` y añadimos a la cabecera la otra condición (la de que la nota esté en el rango correcto). Usamos un `bool`:

```
double nota, media = 0.0;
int total_introducidos = 0;
bool es_nota_correcta = true;

for (i = 0; i < 6 && es_nota_correcta; i++){
    cin >> nota;

    if (0 <= nota && nota <= 10){
        total_introducidos++;
        media = media + nota;
    }
    else
        es_nota_correcta = false;
}

media = media / total_introducidos;
```



Tanto la versión con un bucle `while` de la página 284 como ésta son correctas.

Ejemplo. Compruebe si un número es primo. Lo resolvimos en la página 243:

```
es_primo = true;
divisor = 2;

while (divisor < valor && es_primo){
    if (valor % divisor == 0)
        es_primo = false;
    else
        divisor++;
}
```

Con un for quedaría:

```
es_primo = true;
divisor = 2

for (divisor = 2; divisor < valor && es_primo; divisor++)
    if (valor % divisor == 0)
        es_primo = false;
```

Observe que en la versión con for, la variable `divisor` **siempre** se incrementa, por lo que al terminar el bucle, si el número no es primo, `divisor` será igual al primer divisor de `valor`, más 1.

En los ejemplos anteriores

```
for (i = 0; i < 6 && es_nota_correcta; i++)  
for (divisor = 2; divisor <= tope && es_primo; divisor++)
```

se ha usado dentro del `for` dos condiciones que controlan el bucle:

- ▷ La condición relativa a la variable contadora.
- ▷ Otra condición adicional.

Este código es completamente aceptable en C++.

Consejo: *Limite el uso del bucle `for` en los casos en los que siempre exista:*

- ▷ *Una sentencia de inicialización del contador*
- ▷ *Una condición de continuación que involucre al contador (puede haber otras condiciones **adicionales**)*
- ▷ *Una sentencia final que involucre al contador*



Pero ya puestos, ¿puede usarse entonces, cualquier condición dentro de la cabecera del for? Sí, pero no es recomendable.

Ejemplo. Construya un programa que indique el número de valores que introduce un usuario hasta que se encuentre con un cero (éste no se cuenta)

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

int main(){
    int num_valores, valor;

    cout << "Se contarán el número de valores introducidos";
    cout << "\nIntroduzca 0 para terminar ";

    cin >> valor;
    num_valores = 0;

    while (valor != 0){
        cin >> valor;
        num_valores++;
    }

    cout << "\nEl número de valores introducidos es " << num_valores;
}
```



Lo hacemos ahora con un for:

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

int main(){
    int num_valores, valor;

    cout << "Se contarán el número de valores introducidos";
    cout << "\nIntroduzca 0 para terminar ";

    cin >> valor;

    for (num_valores = 0; valor != 0; num_valores++)
        cin >> valor;

    cout << "\nEl número de valores introducidos es " << num_valores;
}
```

El bucle funciona correctamente pero es un estilo que debemos evitar pues confunde al programador. Si hubiésemos usado otros (menos recomendables) nombres de variables, podríamos tener lo siguiente:

```
for (recorrer = 0; recoger != 0; recorrer++)
```

Y el cerebro de muchos programadores le engañará y le harán creer que está viendo lo siguiente, que es a lo que está acostumbrado:

```
for (recorrer = 0; recorrer != 0; recorrer++)
```

Consejo: Evite la construcción de bucles for en los que la(s) variable(s) que aparece(n) en la condición, no aparece(n) en las otras dos expresiones



Y ya puestos, ¿podemos suprimir algunas expresiones de la cabecera de un bucle `for`? La respuesta es que sí, pero hay que evitarlas SIEMPRE. Oscurecen el código.

Ejemplo. Sume valores leídos desde la entrada por defecto, hasta introducir un cero.

```
int main(){
    int valor, suma;
    cin >> valor;

    for ( ; valor != 0 ; ){
        suma = suma + valor
        cin >> valor;
    }
    .....
}
```



II.3.3. Otras (perniciosas) estructuras de control

Existen otras sentencias en la mayoría de los lenguajes que permiten alterar el flujo normal de un programa.

En concreto, en C++ existen las siguientes sentencias:

`goto` `continue` `break` `exit`



Durante los 60, quedó claro que el uso incontrolado de sentencias de transferencia de control era la principal fuente de problemas para los grupos de desarrollo de software.

Fundamentalmente, el responsable de este problema era la sentencia `goto` que le permite al programador transferir el flujo de control a cualquier punto del programa.

Esta sentencia aumenta considerablemente la complejidad tanto en la legibilidad como en la depuración del código.

Ampliación:

Consulte el libro Code Complete de McConnell, disponible en la biblioteca. En el tema de Estructuras de Control incluye una referencia a un informe en el que se reconoce que un uso inadecuado de un `break`, provocó un apagón telefónico de varias horas en los 90 en NY.



En FP, no se permitirá el uso de ninguna de las sentencias `goto`, `break`, `exit`, `continue`, excepto la sentencia `break` para salir de un `case` dentro de un `switch` (y recuerde lo indicado en la página 174 sobre la fragilidad de la estructura `switch`)



Bibliografía recomendada para este tema:

▷ **A un nivel menor del presentado en las transparencias:**

- Segundo capítulo de Deitel & Deitel
- Capítulos 15 y 16 de McConnell

▷ **A un nivel similar al presentado en las transparencias:**

- Segundo y tercer capítulos de Garrido.
- Capítulos cuarto y quinto de Gaddis.

▷ **A un nivel con más detalles:**

- Capítulos quinto y sexto de Stephen Prata.
- Tercer capítulo de Lafore.

Los autores anteriores presentan primero los bucles junto con la expresiones lógicas y luego los condicionales.

Resúmenes:

Debemos prestar especial atención a los condicionales en los que se asigna un primer valor a alguna variable. Intentaremos garantizar que dicha variable salga siempre del condicional (independientemente de si la condición era verdadera o falsa) con un valor establecido.

Usaremos estructuras condicionales consecutivas cuando los criterios de cada una de ellas sean independientes del resto.

La estructura condicional doble nos permite trabajar con dos condiciones mutuamente excluyentes, comprobando únicamente una de ellas en la parte del `if`. Esto nos permite cumplir el principio de una única vez.



<code>if (cond)</code>		<code>if (cond)</code>
<code>...</code>		<code>...</code>
<code>if (!cond)</code>	→	<code>else</code>
<code>...</code>		<code>...</code>

Utilice las leyes del Álgebra de Boole para simplificar las expresiones lógicas.

La expresión `A || (!A && B)` la sustituiremos por la equivalente a ella: `A || B`

Una situación típica de uso de estructuras condicionales dobles consecutivas, se presenta cuando tenemos que comprobar distintas condiciones independientes entre sí. Dadas n condiciones que dan lugar a n condicionales dobles consecutivos, se pueden presentar 2^n situaciones posibles.



Recordemos que las estructuras condicionales consecutivas se utilizan cuando los criterios de las condiciones son independientes. Por contra, Usaremos los condicionales anidados cuando los criterios sean dependientes entre sí.

<pre>if (c1 && c2 && c3) <accion_A> if (c1 && c2 && !c3) <accion_B> if (c1 && !c2) <accion_C> if (!c1) <accion_D></pre>	→	<pre>if (c1) if (c2) if (c3) <accion_A> else <accion_B> else <accion_C> else <accion_D></pre>
		

Usaremos los condicionales dobles y anidados de forma coherente para no duplicar ni el código de las sentencias ni el de las expresiones lógicas de los condicionales, cumpliendo así el principio de una única vez.

Si debemos comprobar varias condiciones, todas ellas mutuamente excluyentes entre sí, usaremos estructuras condicionales dobles anidadas

Cuando `c1`, `c2` y `c3` sean mutuamente excluyentes:

<pre>if (c1) ... if (c2) ... if (c3) ...</pre>	→	<pre>if (c1) ... else if (c2) ... else if (c3) ...</pre>
		

El compilador no comprueba que cada `case` termina en un `break`. Es por ello, que debemos evitar, en la medida de lo posible, la sentencia `switch` y usar condicionales anidados en su lugar.

El uso de variables intermedias para determinar criterios nos ayuda a no repetir código y facilita la legibilidad de éste. Se les asigna un valor en un bloque y se observa su contenido en otro.

El uso de variables intermedias nos ayuda a separar los bloques de E/S y C. Se les asigna un valor en un bloque y se observa su contenido en otro.

Jamás usaremos un tipo string para detectar un número limitados de alternativas posibles ya que es propenso a errores.

El tipo enumerado puede verse como una extensión del tipo `bool` ya que nos permite manejar más de dos opciones excluyentes.

Al igual que un `bool`, un dato enumerado contendrá un único valor en cada momento. La diferencia está en que con un `bool` sólo tenemos 2 posibilidades y con un enumerado tenemos más (pero no tantas como las 256 de un `char`, por ejemplo).

En el diseño de los bucles siempre hay que comprobar el correcto funcionamiento en los casos extremos (primera y última iteración)

Para no repetir código en los bucles que leen datos, normalmente usaremos la técnica de lectura anticipada:

```
cin >> dato;

while (dato no es último){
    procesar_dato
    cin >> dato;
}
```

A veces también puede ser útil introducir variables lógicas que controlen la condición de terminación de lectura:

```
do{
    cin >> dato;

    test_dato = ¿es el último?

    if (test_dato)
        procesar_dato
}while (test_dato);
```

Evite, en la medida de lo posible, preguntar por algo que sabemos de antemano que sólo va a ser verdadero en la primera iteración.

Debemos garantizar que en algún momento, la condición del bucle se hace falsa.

Tenga especial cuidado dentro del bucle con los condicionales que modifican alguna variable presente en la condición.

En la construcción de bucles con condiciones compuestas, empiece planteando las condiciones simples que la forman. Piense cuándo queremos salir del bucle y conecte adecuadamente dichas condiciones simples, dependiendo de si nos queremos salir cuando todas simultáneamente sean `false` (conectamos con `||`) o cuando cualquiera de ellas sea `false` (conectamos con `&&`)

Diseñe las condiciones compuestas de forma que sean fáciles de leer: en muchas situaciones, nos ayudará introducir variables lógicas a las que se les asignará un valor para salir del bucle cuando se verifique cierta condición de parada.

Si una vez que termina un bucle controlado por varias condiciones, necesitamos saber cuál de ellas hizo que terminase éste, introduciremos variables intermedias para determinarlo. Nunca repetiremos la evaluación de condiciones.

Construya los bucles de forma que no haya que arreglar nada después de su finalización

Usaremos los bucles `for` cuando sepamos, antes de entrar al bucle, el número de iteraciones que se tienen que ejecutar.

Cuando necesitemos puntualmente variables intermedias para realizar nuestros cálculos, será útil definirlas en el ámbito del bloque de instrucciones correspondiente.

Pero hay que tener cuidado si es un bloque `while` ya que pierde el valor en cada iteración.

Esto no ocurre con las variables contadoras del bucle `for`, que recuerdan el valor que tomó en la iteración anterior.

Al diseñar bucles anidados, hay que analizar cuidadosamente las variables que hay que reiniciar antes de entrar a los bucles más internos

Consejo: *En aquellos casos en los que debe asignarle un valor a una variable `bool` dependiendo del resultado de la evaluación de una expresión lógica, utilice directamente la asignación de dicha expresión en vez de un condicional doble.*



Consejo: *En los condicionales que simplemente comprueban si una variable lógica contiene `true`, utilice el formato `if (variable_logica)`. Si se quiere consultar si la variable contiene `false` basta poner `if (!variable_logica)`*



Consejo: *Fomente el uso de los bucles pre-test. Lo usual es que haya algún caso en el que no queramos ejecutar el cuerpo del bucle ni siquiera una vez.*



Consejo: *Evite el uso de los operadores `++` y `--` en la expresión lógica de una sentencia condicional, debido a las sutiles diferencias que hay en su comportamiento, dependiendo de si se usan en formato prefijo o postfijo*



Consejo: *Limite el uso del bucle `for` en los casos en los que siempre exista:*



- ▷ *Una sentencia de inicialización del contador*
- ▷ *Una condición de continuación que involucre al contador (puede haber otras condiciones **adicionales**)*
- ▷ *Una sentencia final que involucre al contador*

Consejo: *Evite la construcción de bucles `for` en los que la(s) variable(s) que aparece(n) en la condición, no aparece(n) en las otras dos expresiones*



Consejo: *Procurad no abusar de este estilo de codificación, es decir, evitad la construcción de bucles `for` sin ninguna sentencia en su interior*



Destaque visualmente el bloque de instrucciones de una estructura condicional.



Diseñe los algoritmos para que sean fácilmente extensibles a situaciones más generales.



Analice siempre con cuidado las tareas a resolver en un problema e intente separar los bloques de código que resuelven cada una de ellas.



Los bloques de código que realizan entradas o salidas de datos (`cin`, `cout`) estarán separados de los bloques que realizan cálculos.



Los algoritmos que realizan una búsqueda, deben salir de ésta en cuanto se haya encontrado el valor. Normalmente, usaremos una variable lógica para controlarlo.

IMPORTANT

Use descripciones de algoritmos que sean **CONCISAS** con una presentación visual agradable y esquemática.

No se hará ningún comentario de aquello que sea obvio.
Más vale poco y bueno que mucho y malo.

Las descripciones serán de un **BLOQUE** completo.

Sólo se usarán comentarios al final de una línea en casos puntuales, para aclarar el código de dicha línea.



No se debe modificar el valor de la variable controladora, ni el valor final dentro del cuerpo del bucle `for`.

En particular, nunca nos saldremos de un bucle `for` asignándole un valor extremo a la variable contadora.



En FP, no se permitirá el uso de ninguna de las sentencias `goto`, `break`, `exit`, `continue`, **excepto la sentencia `break` para salir de un `case` dentro de un `switch`** (y recuerde lo indicado en la página 174 sobre la *fragilidad* de la estructura `switch`)



Principio de Programación:

Sencillez (Simplicity)

***Fomente siempre la sencillez y la legibilidad en la
escritura de código***



Índice alfabético

- álgebra de boole (boolean algebra), 148
- ámbito (scope), 39, 262
- índice de variación, 37
- algoritmo (algorithm), 3
- ascii, 85
- asociatividad (associativity), 52
- biblioteca (library), 14
- bit, 46
- bucle controlado por condición (condition-controlled loop), 214
- bucle controlado por contador (counter controlled loop), 214
- bucle post-test (post-test loop), 214
- bucle pre-test (pre-test loop), 214
- buffer, 98
- byte, 46
- código binario (binary code), 6
- código fuente (source code), 9
- cadena de caracteres (string), 80
- cadena vacía (empty string), 84
- casting, 64
- code point, 87
- codificación (coding), 87
- coma flotante (floating point), 55
- compilador (compiler), 13
- componentes léxicos (tokens), 19
- condición (condition), 121
- conjunto de caracteres (character set), 85
- constante (constant), 33
- cursor (cursor), 98
- dato (data), 3
- decimal codificado en binario (binary-coded decimal), 57
- declaración (definición) (declaration (definition)), 15
- desbordamiento aritmético (arithmetic overflow), 66
- diagrama de flujo (flowchart), 120
- entero (integer), 47
- entrada de datos (data input), 16
- enumerado (enumeration), 208
- errores en tiempo de compilación (compilation error), 21
- errores en tiempo de ejecución (execution error), 22
- errores lógicos (logic errors), 22
- espacio de nombres (namespace), 18

estilo camelcase, 29
estilo snake case, 29
estilo uppercamelcase, 29
estructura condicional (conditional structure), 121
estructura condicional doble (else conditional structure), 137
estructura repetitiva (iteration/loop), 214
estructura secuencial (sequential control flow structure), 121
evaluación en ciclo corto (short-circuit evaluation), 179
evaluación en ciclo largo (eager evaluation), 179
exponente (exponent), 55
expresión (expression), 40
expresiones aritméticas (arithmetic expression), 64

filtro (filter), 216
flujo de control (control flow), 20, 119
función (function), 16, 43

gigo: garbage input, garbage output, 32
google c++ style guide, 30
hardware, 2

identificador (identifier), 15
implementación de un algoritmo (algorithm implementation), 9
indeterminación (undefined), 60

infinito (infinity), 60
iso-10646, 87
iso/iec 8859-1, 86
iteración (iteration), 215

l-value, 41
lógico (boolean), 103
lectura anticipada, 226
lenguaje de programación (programming language), 7
lenguaje ensamblador (assembly language), 7
lenguajes de alto nivel (high level language), 7
leyes de de morgan (de morgan's laws), 148
literal (literal), 33
literales de cadenas de caracteres (string literals), 33
literales de caracteres (character literals), 33
literales enteros (integer literals), 48
literales lógicos (boolean literals), 33
literales numéricos (numeric literals), 33

macro, 60
mantisa (mantissa), 55
mutuamente excluyente (mutually exclusive), 145

número de orden (code point), 85
números mágicos (magic numbers), 35, 116

notación científica (scientific notation), 54
notación infija (infix notation), 42
notación prefija (prefix notation), 42
operador (operator), 16, 43
operador binario (binary operator), 42
operador de asignación (assignment operator), 16
operador de casting (casting operator), 78
operador n-ario (n-ary operator), 42
operador unario (unary operator), 42
página de códigos (code page), 85
parámetros (parameter), 43
precisión (precision), 58
prevalencia de nombre (name hiding), 263
principio de programación - sencillez (programming principle - simplicity), 189, 305
principio de programación - una única vez (programming principle - once and only once), 112, 117
programa (program), 10
programación (programming), 12
programador (programmer), 2
r-value, 41
rango (range), 45
real (real), 54
redondeo (rounding), 56
reglas sintácticas (syntactic rules), 19
salario bruto (gross income), 37
salario neto (net income), 37
salida de datos (data output), 17
sentencia (sentence/statement), 15
sentencia condicional (conditional statement), 122
sentencia condicional doble (else conditional statement), 137
software, 2
tipo de dato (data type), 15
tipos de datos primitivos (primitive data types), 26
transformación de tipo automática (implicit conversion), 64
unicode, 87
usuario (user), 2
valor (value), 16
variables (variables), 33