

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 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	12
I.2. Especificación de programas	13
I.2.1. Organización de un programa	13
I.2.2. Elementos básicos de un lenguaje de programación	18
I.2.2.1. Tokens y reglas sintácticas	18
I.2.2.2. Palabras reservadas	19
I.2.3. Tipos de errores en la programación	20
I.2.4. Cuidando la presentación	22
I.2.4.1. Escritura de código fuente	22
I.2.4.2. Etiquetado de las Entradas/Salidas	23

I.3. Datos y tipos de datos	24
I.3.1. Representación en memoria de datos e instrucciones	24
I.3.2. Datos y tipos de datos	25
I.3.2.1. Declaración de datos	25
I.3.2.2. Literales	29
I.3.2.3. Datos constantes	30
I.3.2.4. Codificando con estilo	35
I.4. Operadores y expresiones	37
I.4.1. Expresiones	37
I.4.2. Terminología en Matemáticas	39
I.4.3. Operadores en Programación	40
I.5. Tipos de datos simples en C++	42
I.5.1. Los tipos de datos enteros	43
I.5.1.1. Representación de los enteros	43
I.5.1.2. Rango de los enteros	44
I.5.1.3. Literales enteros	45
I.5.1.4. Operadores	46
I.5.1.5. Expresiones enteras	48
I.5.2. Los tipos de datos reales	50
I.5.2.1. Literales reales	50
I.5.2.2. Representación de los reales	51

I.5.2.3. Rango y Precisión	53	II.1. Estructura condicional	98
I.5.2.4. Operadores	56	II.1.1. Flujo de control	98
I.5.2.5. Funciones estándar	57	II.1.2. Estructura condicional simple	102
I.5.2.6. Expresiones reales	58	II.1.2.1. Formato	102
I.5.3. Operando con tipos numéricos distintos	60	II.1.2.2. Diagrama de Flujo	105
I.5.3.1. Asignaciones a datos de expresiones de distinto tipo	60	II.1.2.3. Cuestión de estilo	107
I.5.3.2. Expresiones con datos numéricos de distin- to tipo	64	II.1.2.4. Condiciones compuestas	109
I.5.3.3. El operador de casting (Ampliación)	69	II.1.2.5. Estructuras condicionales consecutivas	110
I.5.4. El tipo de dato carácter	71	II.1.3. Estructura condicional doble	114
I.5.4.1. Rango	71	II.1.3.1. Formato	114
I.5.4.2. Literales de carácter	74	II.1.3.2. Variables no asignadas en los condicionales	119
I.5.4.3. Funciones estándar y operadores	76	II.1.3.3. Condiciones mutuamente excluyentes	121
I.5.5. El tipo de dato cadena de caracteres	77	II.1.3.4. Estructuras condicionales dobles consecu- tivas	124
I.5.6. El tipo de dato lógico o booleano	81	II.1.4. Anidamiento de estructuras condicionales	128
I.5.6.1. Rango	81	II.1.5. Estructura condicional múltiple	144
I.5.6.2. Funciones standard y operadores lógicos	81	II.1.6. Programando como profesionales	147
I.5.6.3. Operadores Relacionales	84	II.1.6.1. Diseño de algoritmos fácilmente extensibles	147
I.5.7. Lectura de varios datos	87	II.1.6.2. Descripción de un algoritmo	155
I.6. El principio de una única vez	92	II.1.6.3. Las expresiones lógicas y el principio de una única vez	159
II. Estructuras de Control	97	II.1.6.4. Separación de entradas/salidas y cálculos	165

II.1.6.5. El tipo enumerado y los condicionales . . .	172
II.1.6.6. Simplificación de expresiones compuestas: Álgebra de Boole	179
II.1.7. Algunas cuestiones sobre condicionales	183
II.1.7.1. Cuidado con la comparación entre reales . .	183
II.1.7.2. Evaluación en ciclo corto y en ciclo largo . .	185
II.2. Estructuras repetitivas	186
II.2.1. Bucles controlados por condición: pre-test y post-test	186
II.2.1.1. Formato	186
II.2.1.2. Algunos usos de los bucles	189
II.2.1.3. Bucles con lectura de datos	195
II.2.1.4. Bucles sin fin	200
II.2.2. Programando como profesionales	202
II.2.2.1. Condiciones compuestas	202
II.2.2.2. Bucles que buscan	206
II.2.2.3. Evaluación de expresiones dentro y fuera del bucle	209
II.2.2.4. Bucles que no terminan todas sus tareas . .	211
II.2.2.5. Estilo de codificación	215
II.2.3. Bucles controlador por contador	216
II.2.3.1. Motivación	216
II.2.3.2. Formato	218

II.2.4. Anidamiento de bucles	225
II.3. Particularidades de C++	233
II.3.1. Expresiones y sentencias son similares	233
II.3.1.1. El tipo <code>bool</code> como un tipo entero	233
II.3.1.2. El operador de asignación en expresiones .	235
II.3.1.3. El operador de igualdad en sentencias . . .	236
II.3.1.4. El operador de incremento en expresiones .	237
II.3.2. El bucle <code>for</code> en C++	239
II.3.2.1. Bucles <code>for</code> con cuerpo vacío	239
II.3.2.2. Bucles <code>for</code> con sentencias de incremento in- correctas	239
II.3.2.3. Modificación del contador	240
II.3.2.4. El bucle <code>for</code> como ciclo controlado por con- dición	241
II.3.3. Otras (perniciosas) estructuras de control	249
III. Funciones y Clases	251
III.1. Funciones	252
III.1.1. Fundamentos	252
III.1.1.1. Las funciones realizan una tarea	252
III.1.1.2. Definición	254
III.1.1.3. Parámetros formales y actuales	255
III.1.1.4. Ámbito de un dato. Datos locales	267

III.1.1.5. La Pila	277
III.1.2. El principio de ocultación de información	282
III.1.3. Funciones void	286
III.1.4. Ámbito de un dato (revisión)	290
III.1.5. Parametrización de funciones	296
III.1.6. Programando como profesionales	303
III.1.6.1. Cuestión de estilo	303
III.1.6.2. Diseño de la cabecera de una función	306
III.1.6.3. Precondiciones	312
III.1.6.4. Documentación de una función	313
III.2. Clases	316
III.2.1. Motivación. Clases y Objetos	317
III.2.2. Encapsulación	321
III.2.2.1. Datos miembro	323
III.2.2.2. Métodos	327
III.2.3. Ocultación de información	342
III.2.3.1. Ámbito público y privado	342
III.2.3.2. Datos miembro privados	345
III.2.3.3. Métodos privados	356
III.2.4. Estado inválido de un objeto	363
III.2.5. Constructores	370

III.2.5.1. Definición de constructores	370
III.2.5.2. Sobrecarga de constructores	378
III.2.5.3. Llamadas entre constructores	380
III.2.5.4. Constructores sin parámetros	382
III.2.6. Programando como profesionales	387
III.2.6.1. Datos miembro vs parámetros de los métodos	388
III.2.6.2. Principio de Responsabilidad Única y cohe- sión de una clase	392
III.2.6.3. Funciones vs Clases	398
III.2.6.4. Comprobación y tratamiento de las precon- diciones	403
III.2.7. Registros	411
III.2.7.1. struct y funciones	413
III.2.7.2. Ámbito de un struct	414
III.2.7.3. Inicialización de los campos de un struct	415
III.2.8. Datos miembro constantes	416
III.2.8.1. Constantes a nivel de objeto	417
III.2.8.2. Constantes a nivel de clase	420

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.

Autor: Juan Carlos Cubero.

Sugerencias: por favor, enviar un e-mail a JC.Cubero@decsai.ugr.es

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 \(Programmer\)](#)

1.1.2. 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 representación simbólica de una característica o propiedad de una entidad.

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.

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

- ▷ **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.

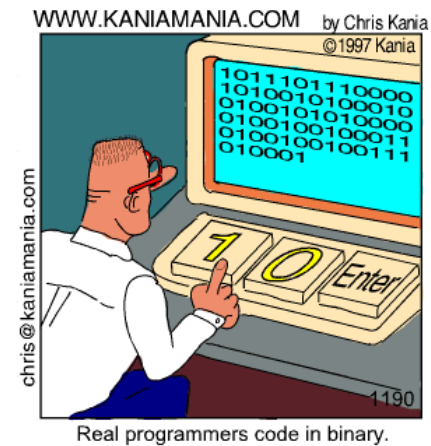
Una vez diseñado el algoritmo, debemos escribir las órdenes que lo constituyen en un lenguaje que entienda el ordenador.

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



I.1.3. Lenguajes de programación

Código binario (Binary code) :



"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
    Cadenal DB 'Hola Mundo.$'
.code
    mov ax, @data
    mov ds, ax
    mov dx, offset Cadenal
    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++1114 (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++?

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.

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 (izda = 1; izda < total_utilizados; izda++){
    a_desplazar = vector[izda];

    for (i = izda; i>0 && a_desplazar < vector[i-1]; i--)
        vector[i] = vector[i-1];

    vector[i] = a_desplazar;
}
```

Estas instrucciones 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.

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/~carlos/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.

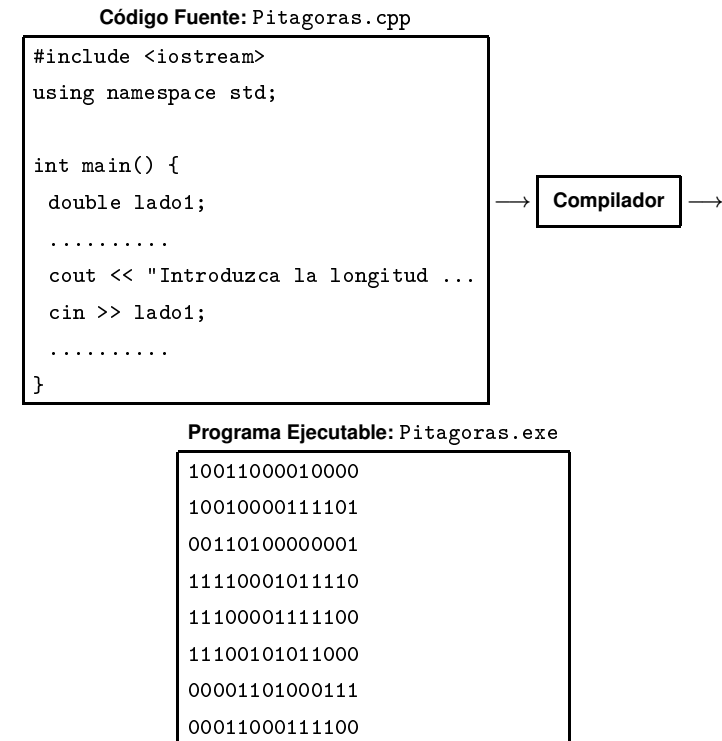
Muchos de los programas que se verán en FP implementarán un único algoritmo.

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**



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, interrumpir 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*

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)

El compilador ofrece distintos *tipos de datos (data types)*, como enteros (`int`), reales (`double`), caracteres (`char`), etc. En una sentencia de *declaración (declaration)*, 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`),

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

Declara tres datos de tipo real que el programador puede usar en el programa.

▷ 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**.

```
cin >> dato;
```

Por ejemplo, al ejecutarse la sentencia

```
cin >> lado1;
```

el programa espera a que el usuario introduzca un valor real (double) desde el teclado (dispositivo de entrada) y, cuando se pul-

sa 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 línea ->\n";  
cout << "\n<- Empiezo en una nueva línea.";
```

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

```
cout << 3.1415927;
```

– Si ponemos un dato, se imprime su contenido.

```
cout << hipotenusa;
```

Podemos usar una única sentencia:

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

Realmente, hay que anteponer el *espacio de nombres (namespace)* `std` en la llamada a `cout` y `cin`:

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

Al haber incluido `using namespace std;` al inicio del programa, ya no es necesario. 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 los usaremos.

Estructura básica de un programa (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.

```
main ; ( == = hipotenusa * /*
```

Pero por ejemplo, ni `i` 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.

```
int dato_entero;
int otra_variable;

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



▷ **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 10 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.

Datos

"Juan Pérez" →

1	0	1	1	..	0	1	1
---	---	---	---	----	---	---	---

75225813 →

1	1	0	1	..	0	0	0
---	---	---	---	----	---	---	---

3.14159 →

0	0	0	1	..	1	1	1
---	---	---	---	----	---	---	---

Instrucciones

Abrir Fichero →

0	0	0	1	..	1	1	1
---	---	---	---	----	---	---	---

Imprimir →

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

Un programa necesitará representar información de diverso tipo (cadenas de caracteres, enteros, reales, etc) 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 III).

Declaración de un dato (data declaration) : Es la sentencia en la que se asigna un nombre a un dato y se le asocia un tipo de dato. El valor que se le puede asignar a un dato depende del tipo de dato con el que es declarado.

Cuando se declara un dato de un tipo primitivo, 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

Podemos declarar varias variables de un mismo tipo en la misma línea. Basta separarlas con coma:

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

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

```
<tipo> <identificador> = <valor_inicial>;  
  
double dato = 4.5;
```

equivale a:

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

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 ñ, 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. No es recomendable usar el nombre de algún identificador usado en las *bibliotecas estándar* (por ejemplo, cout)

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

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

- a) registro1 b) 1registro c) archivo_3 d) main
- e) nombre y direccion f) dirección g) diseño

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

```
/*
    Programa para calcular la retención a aplicar en
    el sueldo de un empleado
*/
#include <iostream>
using namespace std;

int main(){
    double salario_bruto; // Salario bruto, en euros
    double retencion;     // Retención a aplicar, en euros

    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 un dato no asignado:

```
int main(){
    double salario_bruto; // Salario bruto, en euros
    double retencion;     // Retención a aplicar, en euros

    retencion = salario_bruto * 0.18; // salario_bruto indeterminado

    cout << "Retención a aplicar: " << retencion;
}
```



Imprimirá un valor indeterminado.

I.3.2.2. 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', **Literales lógicos (boolean literals)** true, etc.

I.3.2.3. 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. Calculad 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/~carlos/FP/I_circunferencia.cpp

Comparar el anterior código con el siguiente:

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



Ventajas al usar constantes:

- ▷ El nombre dado a la constante (PI) 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 PI (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.

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

Ejercicio. Modificad el ejemplo de la página 28 introduciendo una constante que contenga el valor del porcentaje de la retención (0.18).

```
/*
 Programa para calcular la retención a aplicar en
 el sueldo de un empleado
*/
#include <iostream>
using namespace std;

int main(){
    const double PORCENTAJE_RETENCION = 0.18; // Porcentaje de retención
    double retencion; // Retención a aplicar, en euros
    double salario_bruto; // Salario bruto, en euros

    salario_bruto  PORCENTAJE_RETENCION  retencion
    [?] [0.18] [?]

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

    retencion = salario_bruto * PORCENTAJE_RETENCION;

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

salario_bruto	PORCENTAJE_RETENCION	retencion
32538.0	0.18	4229.94

http://decsai.ugr.es/~carlos/FP/I_retencion.cpp

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.3.2.4. Codificando con estilo

- ▷ 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

- ▷ No se nombrarán dos datos con identificadores 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.

- ▷ Cuando veamos clases, funciones y métodos, éstos se escribirán con la primera letra en mayúscula. Los nombres compuestos se separarán con una mayúscula.



Clase CuentaBancaria, **Método** Ingresa

En el examen, se baja puntos por no seguir las anteriores normas.

IMPORTANT

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
3 + 5
lado1
```

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

```
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 de reales

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 95). Mejor guardamos el resultado de la expresión en una variable y mostramos la variable:

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

Evita 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.
`seno(3), tangente(x), media(valor1, 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:
`seno(3), tangente(x)`
- ▷ **Operador binario (Binary operator)** . Tienes dos argumentos:
`media(valor1, 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. Ejemplos:
`+` (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. Ejemplos:
`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.

A los argumentos de las funciones se les denomina *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.

Ejercicio. Construid un programa en C++ para que lea desde teclado un valor de aceleración y masa de un objeto y calcule la fuerza correspondiente según la segunda ley de Newton:

$$F = m * a$$

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, cuanto 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.

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.

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

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

0000000000110101

- ▷ Dos elementos a combinar: 1, 0
- ▷ r posiciones. Por ejemplo, $r = 16$
- ▷ Se permiten repeticiones e importa el orden
 $0000000000110101 \neq 0000000000110110$
- ▷ Número de datos distintos representables = 2^r

Se conoce como *bit* a la aparición de un valor 0 o 1. Un *byte* es una secuencia de 8 bits.

Esta representación de un entero es válida en cualquier lenguaje de programación.

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 utiliza 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:

$$\left[-\frac{2^{32}}{2}, \frac{2^{32}}{2} - 1 \right] = [-2147483648, 2147483647]$$

```
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:

$$[-9223372036854775808, 9223372036854775807]$$

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

I.5.1.3. Literales enteros

Como ya vimos en la página 29, 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.

I.5.1.4. Operadores

Operadores binarios

+ - * / %

suma, resta, producto, división entera y módulo.

El operador módulo (%) representa el resto de la división entera
Devuelven un entero.

Binarios. Notación infija: `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 → 4`
- ▷ **Truncar desde el dígito menos significativo:** `5734 / 10 → 573`

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; */
```

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

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

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, forzar 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, indicad 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. Leed 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.

Ejemplo. Incrementar el salario en 100 euros y calcular el número de billetes de 500 euros a usar en el pago de dicho salario.

```
int salario, num_bill500;
salario = 43000;
num_bill500 = (salario + 100) / 500; // ok
num_bill500 = salario + 100 / 500;   // Error lógico
```

I.5.2. Los tipos de datos reales

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

▷ Parte entera de $4,56 \rightarrow 4$

▷ Parte real de $4,56 \rightarrow 56$

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 → valor = 4,2001 escala = 10
42001 → valor = 4,2001 escala = 10^4
0,42001 → valor = 4,2001 escala = 10^{-1}

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

En la representación en coma flotante, 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)*

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

Por tanto:

¡Todas las operaciones realizadas con los reales pueden devolver valores que sólo sean aproximados!

IMPORTANT

Especial cuidado tendremos con operaciones del tipo

Repite varias veces

Ir sumándole a una `variable_real` varios valores reales;

ya que los errores de aproximación se irán acumulando.



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 y double (y long double a partir de C++11). 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)
```

[illegible]

Problema de precisión

```
valor_real = 0.100000000000000000009;  
// Almacena: 0.100000000000000001
```



Problema de precisión

```
valor_real = 1.000000000000000000009;  
// Almacena: 1.0
```



Problema de precisión

```
valor_real = 10000000001.0000000000000004;  
// 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.

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

Representaremos infinito por `INF` y la indeterminación por `NaN`, pero hay que destacar que no son literales que puedan usarse en el código.

Las operaciones numéricas con infinito son las usuales en Matemáticas (1.0/INF 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 INF
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 INF
valor_real = 1.0 / valor_real;         // Almacena 0.0
```

I.5.2.4. 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
```

I.5.2.5. 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`

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

Todos los anteriores son unarios excepto `pow`, que es binario (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:

Observad 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.5.2.6. Expresiones reales

Son expresiones cuyo resultado es un número real.

`sqrt(real)` es una expresión real

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

En general, diremos que las *expresiones aritméticas* (*arithmetic expression*) o *numéricas* son las expresiones o bien enteras o bien reales.

Precedencia de operadores en las expresiones reales:

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

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



Ejemplo. Construid una expresión para calcular la siguiente fórmula:

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

`-b+sqrt(b*b-4.0*a*c)/2.0*a`

Error lógico

`((-b)+(sqrt((b*b) - (4.0*a)*c)))/(2.0*a)`

Difícil de leer

`(-b + sqrt(b*b - 4.0*a*c)) / (2.0*a)`

Correcto

Ejercicio. Construid un programa para calcular la posición de un objeto que sigue un movimiento rectilíneo uniforme. La posición se calcula aplicando la siguiente fórmula:

$$x_o + vt$$

dónde x_o es la posición inicial, v la velocidad y t el tiempo transcurrido. Suponed que los tres datos son reales.

Ejercicio. Construid una expresión para calcular la distancia euclídea entre dos puntos del plano $P1 = (x_1, y_1)$, $P2 = (x_2, y_2)$. Usad 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

I.5.3.1. Asignaciones a datos de expresiones de distinto tipo

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

En primer lugar se evalúa la parte derecha de la asignación.

En segundo lugar, se realiza la asignación. 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 (casting)* de la expresión de la derecha al tipo de dato de la parte izquierda de la asignación.

Esta transformación es temporal (mientras se evalúa la expresión)

Si el resultado de la expresión no cabe en la parte izquierda, se produce un error lógico denominado *desbordamiento aritmético (arithmetic overflow)*. El valor asignado será un valor indeterminado.

Veamos varios casos:

- ▷ Como cabría esperar, a un dato numérico de tipo *grande* se le puede asignar cualquier expresión numérica de tipo *pequeño*.

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

```
// grande = chico; Sin problemas.
```

```
chico = 5;      // 5 es un literal int
               // int = int
```

```
grande = chico;    // chico int -> chico long long
                  // long long = long long
```

```
cout << chico;    // chico sigue siendo int
```

La transformación de `int` a `long long` es inmediata:

```
5 int:
```

000000000000000000000000000000000000101

5 long long:

[illegible]

Otro ejemplo:

```
double real;
```

```
int entero;
```

```
entero = 5;
```

```
real    = entero;    // 5 int -> 5 double
                        // double = double
```

```
cout << entero;    // entero sigue siendo int
                  // Imprime 5 y no 5.0
```

- ▷ 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;    // 36000000000000 no cabe en un int
                  // Desbordamiento.
                  // chico = -415875072
                  // int = int
```

- ▷ 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
```

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.

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

I.5.3.2. Expresiones con datos numéricos de distinto tipo

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 de la expresión.

Esta transformación es temporal (mientras se evalúa la expresión)

Ejemplo. Calcular 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
```

Si en una expresión todos los datos son del mismo tipo, el compilador no realiza casting.

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

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

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



La expresión `edad1 + edad2` es entera y devuelve 15. Por tanto, 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_tmp;

edad1_tmp = edad1;
media      = (edad1_tmp + 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.

▷ 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, cabe la posibilidad de reconsiderar la decisión con respecto a los tipos asociados y cambiarlos si la semántica de las variables 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.5.3.3** (página 69)

▷ 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
```



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`)

Observad 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 (recordad lo visto en la página 41)

Posibles soluciones: Las mismas que vimos en el ejemplo anterior (las tres primeras, porque no hay literales involucrados en la expresión)

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
```

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.5.3.3. 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 67:

```
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
```



¿Por qué no es correcto lo siguiente?

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

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

// grande = 304084036
```



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 carácter

I.5.4.1. Rango

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), tipo de moneda, etc.

Los caracteres que pueden usarse son los permitidos en la plataforma en la que se ejecuta el programa. Para representar dichos caracteres se pueden usar distintos tipos de *codificación (coding)*. El estándar aceptado actualmente es *Unicode* (multi-lenguaje, multi-plataforma). Éste establece los caracteres que pueden representarse (incluye caracteres de idiomas como español, chino, árabe, etc), cómo hacerlo (permite tres tamaños distintos: 8, 16 y 32 bits) y asigna un número de orden a cada uno de ellos.

Los primeros 256 caracteres de Unicode coinciden con la codificación antigua denominada *ASCII Extendido (Extended ASCII)*. De éstos, los primeros 32 no son imprimibles. Los caracteres con acentos especiales usados en Europa Occidental (incluida España) están incluidos en el ASCII extendido.

0	<NUL>	32	<SPC>	64	@	96	`	128	À	160	†	192	¿	224	‡
1	<SOH>	33	!	65	A	97	a	129	Á	161	°	193	¡	225	·
2	<STX>	34	"	66	B	98	b	130	Ç	162	¢	194	í	226	,
3	<ETX>	35	#	67	C	99	c	131	É	163	£	195	√	227	„
4	<EOT>	36	\$	68	D	100	d	132	Ñ	164	§	196	ƒ	228	‰
5	<ENQ>	37	%	69	E	101	e	133	Ó	165	•	197	≈	229	Â
6	<ACK>	38	&	70	F	102	f	134	Ü	166	¶	198	Δ	230	Ê
7	<BEL>	39	'	71	G	103	g	135	á	167	ß	199	«	231	Á
8	<BS>	40	(72	H	104	h	136	à	168	®	200	»	232	È
9	<TAB>	41)	73	I	105	i	137	â	169	©	201	...	233	É
10	<LF>	42	*	74	J	106	j	138	ä	170	™	202		234	Î
11	<VT>	43	+	75	K	107	k	139	ã	171	ˆ	203	À	235	Ï
12	<FF>	44	,	76	L	108	l	140	ä	172	˜	204	Ã	236	Ï
13	<CR>	45	-	77	M	109	m	141	ç	173	#	205	Ö	237	Ì
14	<SO>	46	.	78	N	110	n	142	é	174	Æ	206	œ	238	Ó
15	<SI>	47	/	79	O	111	o	143	è	175	Ø	207	ø	239	Ô
16	<DLE>	48	0	80	P	112	p	144	ê	176	∞	208	—	240	Ⓜ
17	<DC1>	49	1	81	Q	113	q	145	ë	177	±	209	—	241	Ò
18	<DC2>	50	2	82	R	114	r	146	í	178	≤	210	"	242	Ú
19	<DC3>	51	3	83	S	115	s	147	ì	179	≥	211	'	243	Û
20	<DC4>	52	4	84	T	116	t	148	î	180	¥	212	‘	244	Ü
21	<NAK>	53	5	85	U	117	u	149	ï	181	μ	213	’	245	ı
22	<SYN>	54	6	86	V	118	v	150	ñ	182	ð	214	÷	246	ˆ
23	<ETB>	55	7	87	W	119	w	151	ó	183	Σ	215	◊	247	˜
24	<CAN>	56	8	88	X	120	x	152	ô	184	Π	216	ÿ	248	˘
25		57	9	89	Y	121	y	153	õ	185	π	217	ÿ	249	˙
26	<SUB>	58	:	90	Z	122	z	154	ö	186	ƒ	218	/	250	˚
27	<ESC>	59	;	91	[123	{	155	ø	187	ª	219	€	251	°
28	<FS>	60	<	92	\	124		156	ú	188	º	220	<	252	ˆ
29	<GS>	61	=	93]	125	}	157	ù	189	Ω	221	>	253	˜
30	<RS>	62	>	94	^	126	~	158	û	190	æ	222	fi	254	˘
31	<US>	63	?	95	_	127		159	ü	191	ø	223	fi	255	˙

C++ permite asignar a cualquier entero uno de estos caracteres, encerrándolo entre comillas simples. El valor que el entero almacena es el número de orden de dicho 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
```

Si nos restringimos a los 256 primeros caracteres, sólo necesitamos 1 byte para representarlo. Por eso, C++ ofrece el tipo de dato `char`:

El tipo de dato `char` es un entero pequeño sin signo con rango $\{0 \dots 255\}$

```
char letra_piso;

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

Además, el recurso `cout` se ha programado de la siguiente forma:

- ▷ Si se pasa a `cout` un dato de tipo `char`, imprime el carácter correspondiente al entero almacenado.
- ▷ Si se pasa a `cout` un dato de tipo entero (distinto a `char`), imprime el entero almacenado.

```
int entero = 'A';
char letra = 'A';

cout << entero; // Imprime 65
cout << letra;  // Imprime A
```

I.5.4.2. Literales de carácter

Son tokens formados por:

- ▷ Un único carácter encerrado entre comillas simples:

`'!' 'A' 'a' '5' 'ñ'`

Observad que `'5'` es un literal de carácter y `5` es un literal entero

¡Cuidado!: `'cinco'` o `'11'` no son literales de carácter.

Observad la diferencia:

```
double r;
char letra;

letra = 'r'; // literal de carácter 'r'
r = 23.2;    // variable real r
```

- ▷ O bien una *secuencia de escape*, es decir, el símbolo `\` seguido de otro símbolo, como por ejemplo:

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

Las secuencias de escape también deben ir entre comillas simples, por ejemplo, `'\n'`, `'\t'`, etc.

```
#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 querremos hacer.



I.5.4.3. 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;

    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 entero de orden correspondiente:

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

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

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

¿Qué imprimiría la sentencia `cout << 'A'+1`? No imprime `'B'` como cabría esperar sino `66` ya que `1` es un literal entero y por tanto de tipo `int`. Un `int` es más grande que un `char`, por lo que `'A'+1` es un `int`.

1.5.5. 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 presentes en `cout`

```
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 cuales 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
```



1.5.6. 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`.

1.5.6.1. 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`

1.5.6.2. Funciones standard 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, INF 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. Declarad dos variables de tipo `bool`, `es_joven` y `es_varon`. Asignadles cualquier valor. Declarad otra variable `es_varon_viejo` y asignadle el valor correcto usando las variables anteriores y los operadores lógicos.

I.5.6.3. 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 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: *Simplificad las expresiones lógicas, para así aumentar su legibilidad.*



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

1.5.7. 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;
```

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

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 que previamente hubiese:

```
char caracter;
```

<code>└─ a 1 2 3</code> <code>↑</code>	<code>cin >> caracter;</code> <code>caracter = 'a';</code>	<code>1 2 3</code> <code>↑</code>
---	---	--------------------------------------

Si queremos leer los separadores en una variable de tipo `char` debemos usar `cin.get()`:

<code>└─ a 1 2 3</code> <code>↑</code>	<code>caracter = cin.get();</code> <code>caracter = ' ';</code>	<code>a 1 2 3</code> <code>↑</code>
<code>a 1 2 3</code> <code>↑</code>	<code>caracter = cin.get();</code> <code>caracter = 'a';</code>	<code>1 2 3</code> <code>↑</code>

Lo mismo ocurre si hubiese un carácter de nueva línea:

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

Ampliación:

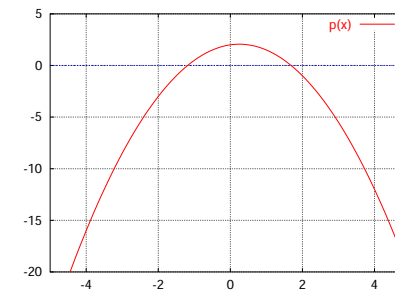
Para leer una cadena de caracteres (`string`) podemos usar la función `getline` de la biblioteca `string`. Permite leer caracteres hasta llegar a un terminador que, por defecto, es el carácter de nueva línea `'\n'`. La cadena a leer se pasa como un parámetro por referencia a la función. Este tipo de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



I.6. El principio de una única vez

Ejemplo. Calcular 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 $\sqrt{b^2 + 4ac}$. 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: $b^2 - 4ac$, por lo que tendremos que cambiar dos líneas.

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/~carlos/FP/I_ecuacion_segundo_grado.cpp

Nota:

Observad 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:

Jamás ha de repetirse código

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. Ésto 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 Deitel & Deitel
 - Primer capítulo de Garrido.
- ▷ **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.

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 de programación en la construcción de las estructuras condicionales y repetitivas.

Autor: Juan Carlos Cubero.

Sugerencias: por favor, enviar un e-mail a JC.Cubero@decsai.ugr.es

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, recordad el ejemplo de la página 94 (no numeramos las sentencias de declaración de los datos):

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

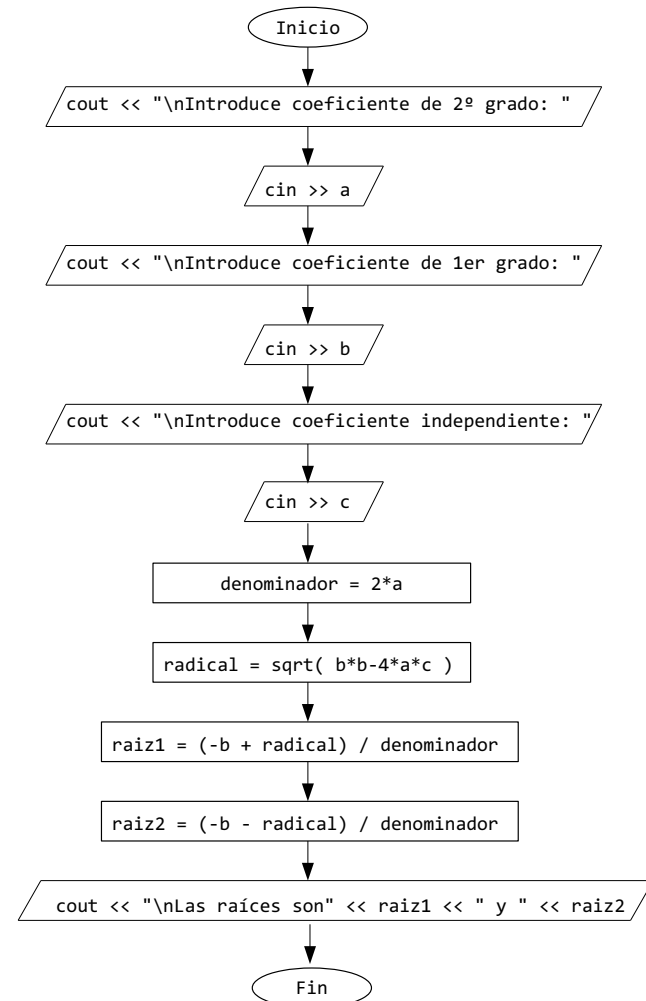
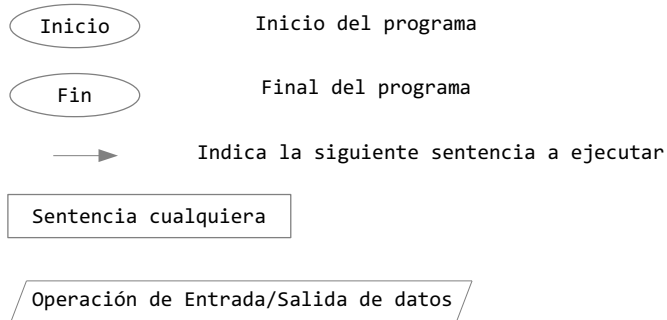
1   cout << "\nIntroduce coeficiente de 2º grado: ";
2   cin >> a;
3   cout << "\nIntroduce coeficiente de 1er grado: ";
4   cin >> b;
5   cout << "\nIntroduce coeficiente independiente: ";
6   cin >> c;

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 incluida en el condicional (incluida la llave cerrada, en su caso), forma una única sentencia, denominada *sentencia condicional (conditional statement)*.

Ejemplo. Leer un número e imprimir "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. En un programa de gestión de notas, subir medio punto a los que han sacado más de 4.5 puntos en el examen escrito.

```
double nota_escrito;

cin >> nota_escrito;

if (nota_escrito >= 4.5)
    nota_escrito = nota_escrito + 0.5;
```

En el caso de que la nota escrita sea menor de 4.5, simplemente no se aplicará la subida. La variable `nota_escrito` se quedará con el valor que tenía.

Ejemplo. Continuando el ejemplo de la página 98

```
#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;
    }
}
```

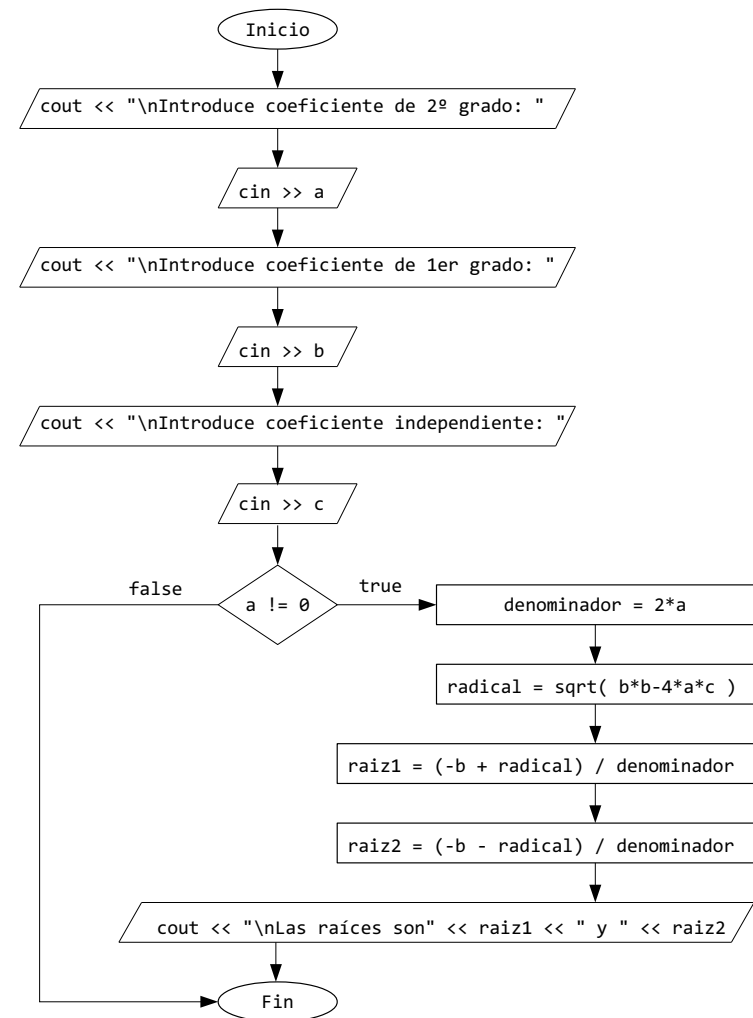


II.1.2.2. Diagrama de Flujo

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



El ejemplo de la ecuación de segundo grado quedaría:



II.1.2.3. 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;
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;
}
```

Línea en blanco antes del condicional

Línea en blanco después del condicional

Bloque if tabulado 3 espacios

Llave cerrada (sin tabulación)

Destacad visualmente el bloque de instrucciones de una estructura condicional.

No seguir estas normas baja puntos en el examen.

IMPORTANT

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;
```

II.1.2.4. Condiciones compuestas

En muchos casos tendremos que utilizar condiciones compuestas:

Ejemplo. Comprobad 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";
```

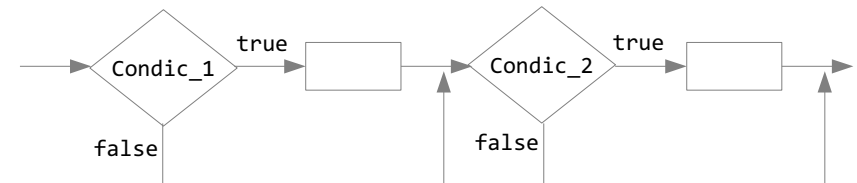
Ejercicio. Subir medio punto al examen escrito de aquellos alumnos que hayan tenido una puntuación entre 4.5 y 8.

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

II.1.2.5. Estructuras condicionales consecutivas

¿Qué ocurre si en el código hay dos estructuras condicionales simples consecutivas? Si las dos condiciones son verdaderas, se ejecutarán los dos bloques de instrucciones correspondientes.

```
if (Condic_1)
    .....
if (Condic_2)
    .....
```



Ejemplo. Comprobad si un número es par o impar y múltiplo de 13 o no.

```
int entero;
cin >> entero;

if (entero % 2 == 0)
    cout << "\n" << entero << " es par";

if (entero % 13 == 0)
    cout << "\n" << entero << " es múltiplo de 13";
```

Dependiendo del valor de `entero`, se pueden imprimir ambos mensajes, uno sólo de ellos o ninguno.

Ejemplo. Comprobad si una persona es mayor de edad y si tiene más de 190 cm de altura.

```
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.

Ejemplo. En un programa de ventas, si la cantidad vendida es mayor de 100 unidades, se le aplica un descuento del 3%. Por otra parte, si el precio final de la venta es mayor de 700 euros, se aplica un descuento del 2%. Ambos descuentos son acumulables.

Para aplicar un descuento del 2%, basta hacer lo siguiente:

```
cantidad = cantidad - cantidad * 2 / 100.0;
```

o directamente:

```
cantidad = cantidad * (1 - 0.02);
```

Si aplicamos una subida, pondríamos $(1 + 0.02)$. En general, el número resultante (0.98, 1.02) se le denomina *índice de variación*.

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

int main(){
    // IV = INDICE_VARIACION

    const int MINIMO_UNIDADES_PARA_DESCUENTO = 100;
    const double MINIMO_VENTA_GRANDE_PARA_DESCUENTO = 700.0;
    const double IV_DESCUENTO_POR_UNIDADES_VENDIDAS = 1 - 0.03;
    const double IV_DESCUENTO_POR_VENTA_GRANDE = 1 - 0.02;
    double precio_unitario, total_venta;
    int unidades_vendidas;

    cout << "Precio unitario: ";
    cin >> precio_unitario;
    cout << "\nUnidades vendidas: ";
    cin >> unidades_vendidas;
```

```
if (unidades_vendidas >= MINIMO_UNIDADES_PARA_DESCUENTO)
    precio_unitario = precio_unitario *
        IV_DESCUENTO_POR_UNIDADES_VENDIDAS;

total_venta = precio_unitario * unidades_vendidas;

if (total_venta >= MINIMO_VENTA_GRANDE_PARA_DESCUENTO)
    total_venta = total_venta *
        IV_DESCUENTO_POR_VENTA_GRANDE;

cout << "\nTotal venta: " << total_venta;
}
```

Dependiendo del número de unidades vendidas y del precio de éstas, podrá aplicarse uno de los dos descuentos y el otro no, los dos, o ninguno.

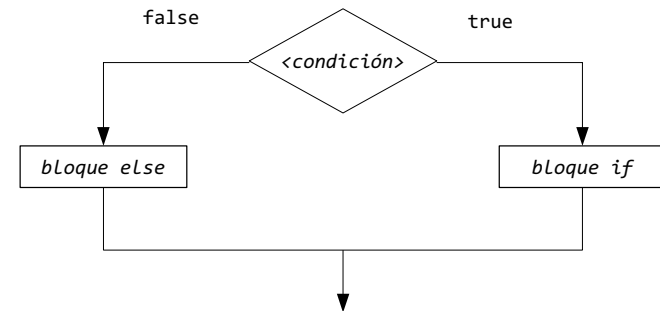
http://decsai.ugr.es/~carlos/FP/II_ventas.cpp

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*) :

```
if <condición>
    <bloque if>;
else
    <bloque else>;
```



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. Comprobad si un número es par (continuación)

```
cin >> entero;

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

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

Ejemplo. Comprobad si un número real está dentro de un intervalo cerrado [inferior, superior] (continuación)

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

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;

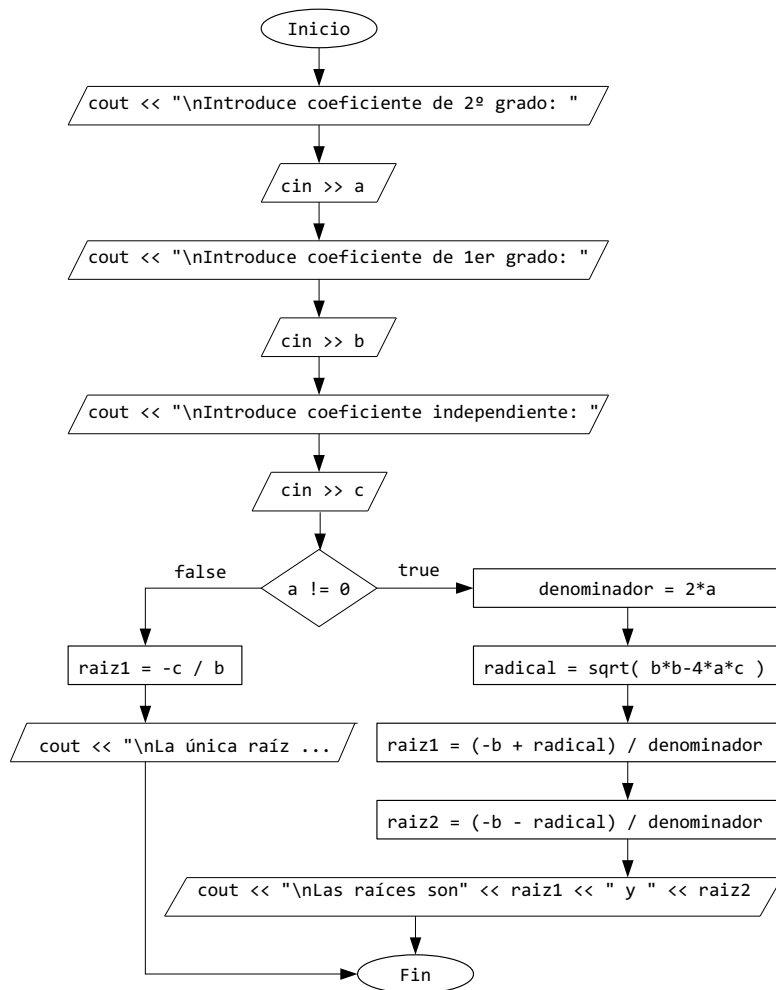
    cout << "\nIntroduce coeficiente de 2º grado: ";
    cin >> a;
    cout << "\nIntroduce coeficiente de 1er grado: ";
    cin >> b;
    cout << "\nIntroduce coeficiente independiente: ";
    cin >> c;

    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. En el ejemplo de la página 103 se subía la nota a los que tenían más de 4.5. ¿Qué ocurre si la nota es menor?

```

cin >> nota_escrito;

if (nota_escrito >= 4.5)
    nota_escrito = nota_escrito + 0.5;
else
    nota_escrito = nota_escrito;
  
```



Obviamente, no es necesario el `else`. Si la nota es menor, la variable `nota_escrito` debe quedarse con el valor que tenía. La sentencia `nota_escrito = nota_escrito;` no sirve de nada ya que no modifica el antiguo valor de la variable.

En definitiva, en este ejemplo, no es necesaria una estructura condicional doble sino sencilla, por lo que nos quedamos con la versión de la página 103.

Lo mismo ocurre en el ejemplo de las ventas de la página 112. No hay que incluir ningún `else`.

II.1.3.2. Variables no asignadas en los condicionales

Supongamos una variable sin 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. En el ejemplo del número par, en vez de imprimir un mensaje, asignamos un valor a una variable de tipo `bool`:

```
int entero;
bool es_par_entero;

cin >> entero;

if (entero % 2 == 0)
    es_par_entero = true;

if (es_par_entero)
    .....
```



Este código puede producir errores lógicos ya que el valor de la variable `es_par_entero` es indeterminado en el caso de que `entero` sea impar.
Solución:

```
int entero;
bool es_par_entero;

cin >> entero;

if (entero % 2 == 0)
    es_par_entero = true;
else
    es_par_entero = false;
```

Y mejor si sustituimos el condicional doble por:

```
int entero;
bool es_par_entero;

cin >> entero;

es_par_entero = entero % 2 == 0;
```

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.

Ejercicio. Siguiendo el ejemplo del intervalo cerrado [inferior, superior], asignad convenientemente un valor a una variable lógica `pertenece_al_intervalo`.

Ejercicio. Leed dos variables enteras `a` y `b` y asignad a una variable `max` el máximo de ambas.

Ejemplo. En el ejemplo de la ecuación de segundo grado, si la variable `a` es igual a cero, sólo se le asigna un valor a `raiz1` pero no a `raiz2`, por ejemplo. Como el programa termina con el condicional, dichas variables no se usan posteriormente, por lo que no tendríamos que cambiar el código. Sin embargo, si no fuese así, sí habría que modificarlo convenientemente. Posteriormente se verá una versión completa de este ejemplo.

II.1.3.3. Condiciones mutuamente excluyentes

Ejemplo. El ejemplo del número par 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.

Utilizaremos una estructura condicional doble cuando tengamos que trabajar con dos condiciones mutuamente excluyentes. Únicamente tendremos que poner en la expresión lógica del `if` una de las dos condiciones

<code>if (entero % 2 == 0)</code>	<code>if (edad >= 18)</code>	<code>if (a != 0)</code>	
.....	
<code>else</code>	<code>else</code>	<code>else</code>	
.....	

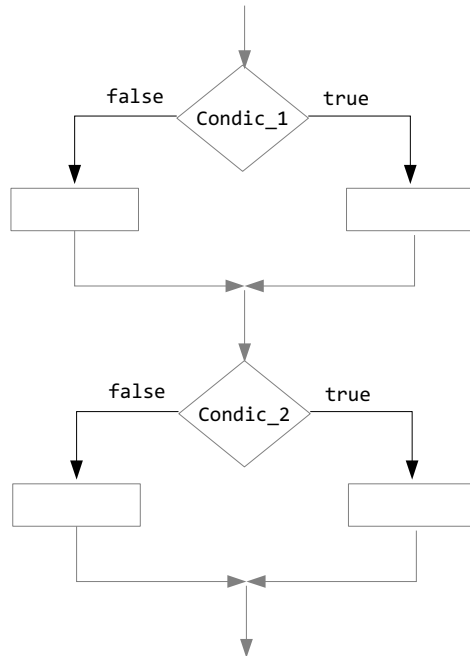
<code>if (entero % 2 == 0)</code>	<code>if (edad >= 18)</code>	<code>if (a != 0)</code>	
.....	
<code>if (entero % 2 != 0)</code>	<code>if (edad < 18)</code>	<code>if (a == 0)</code>	
.....	

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 correspondan: o ninguno, o uno de ellos, o los dos.

```
if (Condic_1)
    .....
else
    .....

if (Condic_2)
    .....
else
    .....
```



Ejemplo. Leer un entero y decir si es par y si es múltiplo de 13. Observad que se pueden dar las cuatro combinaciones. Por ejemplo, 26 es par y múltiplo de 13, 12 es par pero no múltiplo de 13, 39 es impar y múltiplo de 13 y 41 es impar pero no es múltiplo de 13.

```
int entero;
cin >> entero;

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

if (entero % 13 == 0)
    cout << "\n" << entero << " es múltiplo de 13";
else
    cout << "\n" << entero << " no es múltiplo de 13";
```

Siempre se imprimirán dos mensajes. Uno relativo a la condición de ser par o impar y el otro relativo a ser múltiplo o no de 13.

Ejemplo. Leer la edad y la altura de una persona y decir 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.

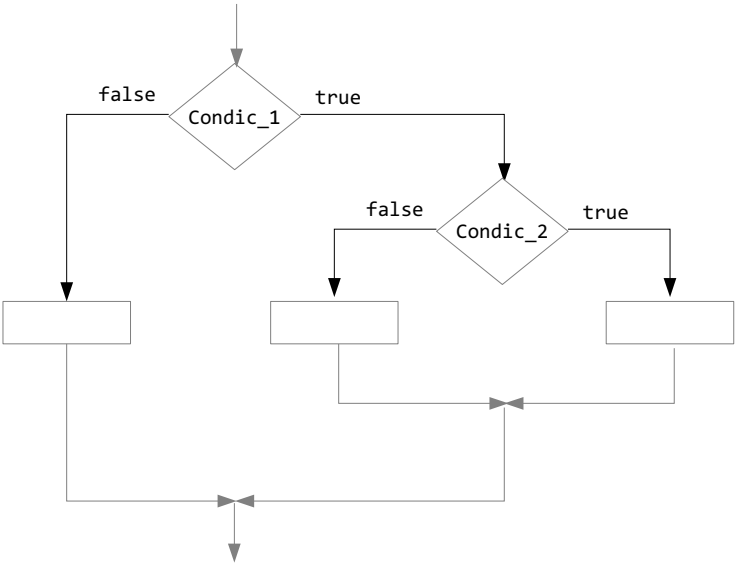
Una situación típica de uso de estructuras condicionales dobles consecutivas, se presenta cuando tenemos que comprobar distintas condiciones independientes entre sí. Dadas n condiciones que dan lugar a n condicionales dobles consecutivos, se pueden presentar 2^n situaciones posibles.

II.1.4. Anidamiento de estructuras condicionales

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){
        .....
    }
    else{
        .....
    }
}
else{
    .....
}
```



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_5	false	da igual	true

Ejemplo. Retomamos el ejemplo de subir la nota de la página 103:

```
double nota_escrito;

cin >> nota_escrito;

if (nota_escrito >= 4.5)
    nota_escrito = nota_escrito + 0.5;
```

Debemos tener en cuenta que la nota no sea nunca mayor de 10. Después de realizar la subida, truncamos a 10 si es necesario. Usamos un condicional anidado:

```
double nota_escrito;

cin >> nota_escrito;

if (nota_escrito >= 4.5){
    nota_escrito = nota_escrito + 0.5;

    if (nota_escrito > 10)
        nota_escrito = 10;
}
```

Observad que al haber más de una instrucción en el bloque if, debemos introducir llaves.

Ejemplo. El el ejemplo de la página 126 queríamos ver si un número era par o impar y múltiplo o no de 13. Cambiamos el ejercicio para comprobar si un número es par, en cuyo caso, queremos ver si es múltiplo de 13. Si no es par, no comprobamos nada más.

Tenemos tres situaciones posibles: par múltiplo de 13, par no múltiplo de 13, impar.

```
cin >> entero;

if (entero % 2 == 0){
    cout << "\n" << entero << " es par";

    if (entero % 13 == 0)
        cout << "\n" << entero << " es múltiplo de 13";
    else
        cout << "\n" << entero << " no es múltiplo de 13";
}
else
    cout << "\n" << entero << " es impar";
```

Si es par, se imprimirán dos mensajes. Uno diciendo que es par y el otro diciendo si es múltiplo o no de 13. Si es impar, sólo se imprimirá un mensaje (diciendo que es impar)

Ejemplo. Sobre el ejemplo anterior, lo cambiamos para comprobar si es múltiplo de 13, sólo cuando sea impar. Si es par, no comprobamos nada más.

Tenemos tres situaciones posibles: par, impar múltiplo de 13, impar no múltiplo de 13.

```
cin >> entero;

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

    if (entero % 13 == 0)
        cout << "\n" << entero << " es múltiplo de 13";
    else
        cout << "\n" << entero << " no es múltiplo de 13";
}
```

Si es impar, se imprimirán dos mensajes. Uno diciendo que es impar y el otro diciendo si es múltiplo o no de 13. Si es par, sólo se imprimirá un mensaje (diciendo que es par)

Ejemplo. Modificamos el ejemplo de la edad y la altura de una persona visto en la página 127. Si es mayor de edad, decimos si es alto o no. Si es menor de edad no hacemos nada más.

Tenemos tres situaciones posibles: mayor de edad alto, mayor de edad no alto, menor de edad.

```
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
    cout << "\nEs menor de edad";
```

Ejemplo. Cambiamos el ejemplo anterior 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.

Tenemos cuatro situaciones 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";

    if (altura >= 190)
        cout << "\nEs alto/a";
    else
        cout << "\nNo es alto/a";
}
else{
    cout << "\nEs menor de edad";

    if (altura >= 175)
        cout << "\nEs alto/a";
    else
        cout << "\nNo es alto/a";
}
```



Observad que si el umbral de la altura fuese el mismo para cualquier edad, la solución es la que vimos en la página 127.

Ejemplo. Refinamos el programa para el cálculo de las raíces de una ecuación de segundo grado visto en la página 116 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.";
}
```



Ejemplo. Reescribid el siguiente ejemplo utilizando únicamente estructuras condicionales no anidadas:

```
if (A>B)
    if (B<=C)
        if (C!=D)
            <S1>;
        else
            <S2>;
    else
        <S3>;
else
    <S4>;
```

↓

```
if ((A>B) && (B<=C) && (C!=D))
    <S1>;
if ((A>B) && (B<=C) && (C==D))
    <S2>;
if ((A>B) && (B>C))
    <S3>;
if (A<=B)
    <S4>;
```

Preferimos el primero ya que realiza menos comprobaciones, y sobre todo porque no duplica código (el de las condiciones) y es, por tanto, menos propenso a errores de escritura o ante posibles cambios futuros.

Ejemplo. Se quiere establecer una bonificación b en la cotización de la Seguridad Social de los trabajadores según la edad (e) y el número de meses trabajados (m) según el siguiente criterio:

- ▷ Si $18 \leq e \leq 35$ y $m \leq 3 \Rightarrow b = 8\%$
- ▷ Si $18 \leq e \leq 35$ y $m > 3 \Rightarrow b = 4\%$
- ▷ Si $e > 35$ y $m \leq 6 \Rightarrow b = 5\%$
- ▷ Si $e > 35$ y $m > 6 \Rightarrow b = 3\%$
- ▷ Si $16 \leq e < 18 \Rightarrow b = 2\%$

Versión sin anidar 😞:

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

```
int main(){
    int edad;
    int meses_trabajados;
    double bonificacion;

    cout << "\nIntroduzca la edad del trabajador";
    cin >> edad;
    cout << "\nIntroduzca el número de meses trabajados";
    cin >> meses_trabajados;
```

// Repetimos código:

```
if (18 <= edad && edad <= 35 && meses_trabajados <= 3)
    bonificacion = 0.08;

if (18 <= edad && edad <= 35 && meses_trabajados > 3)
    bonificacion = 0.04;
```



```
if (35 < edad && meses_trabajados <= 6)
    bonificacion = 0.05;

if (35 < edad && meses_trabajados > 6)
    bonificacion = 0.03;

if (16 <= edad && edad < 18)
    bonificacion = 0.02;

.....
}
```

La expresión `18 <= edad` (por ejemplo) se repite varias veces, por lo que el código es propenso a errores y difícil de actualizar. La versión anidando

no repite el código de las expresiones lógicas relacionales 😊:

```
if (16 <= edad && edad < 18)
    bonificacion = 0.02;
else{
    if (edad <= 35)
        if (meses_trabajados <= 3)
            bonificacion = 0.08;
        else
            bonificacion = 0.04;
    else
        if (meses_trabajados <= 6)
            bonificacion = 0.05;
        else
            bonificacion = 0.03;
}
```



Nota:

Hemos supuesto que el valor de `edad` es correcto (entre 16 y 65, que es la edad laboral) En el caso de que no fuese así, se podría asignar a la bonificación un valor extremo (infinito o indeterminación)

Ejemplo. Menú de operaciones.

```
#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 = 0.0;
        resultado = resultado / resultado;    // NaN
    }

    cout << "\nResultado = " << resultado;
}
```



143

Las condiciones `opcion == 'S'`, `opcion == 'R'`, etc, son mutuamente excluyentes entre sí. Por lo tanto, para evitar evaluar innecesariamente las condiciones lógicas posteriores a la primera que sea `true`, en vez de usar estructuras condicionales simples consecutivas, 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 = 0.0;
            resultado = resultado / resultado;    // NaN
        }
}
```

Si debemos comprobar varias condiciones, todas ellas mutuamente excluyentes entre sí, usaremos estructuras condicionales dobles anidadas

Una forma también válida de tabular:

```
if (opcion == 'S')
    resultado = dato1 + dato2;
else if (opcion == 'R')
    resultado = dato1 - dato2;
else if (opcion == 'M')
    resultado = dato1 * dato2;
else{
    resultado = 0.0;
    resultado = resultado / resultado;    // NaN
}
```



http://decsai.ugr.es/~carlos/FP/II_menu_operaciones.cpp

II.1.5. Estructura condicional múltiple

Recordad el ejemplo de lectura de dos enteros y una opción de la página 143. 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> ó <constante2> 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. Se suele colocar 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 = 0.0;
    resultado = resultado / 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 = 0.0;
        resultado = resultado / 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 = 0.0;
        resultado = resultado / resultado;    // NaN
}
```

No olvide incluir los `break` en las opciones correspondientes de un `switch`.

II.1.6. Programando como profesionales

II.1.6.1. Diseño de algoritmos fácilmente extensibles

Ejemplo. Calculad el mayor de tres números a, b y c (primera aproximación).

Algoritmo: Mayor de tres números. Versión 1

▷ **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))
```

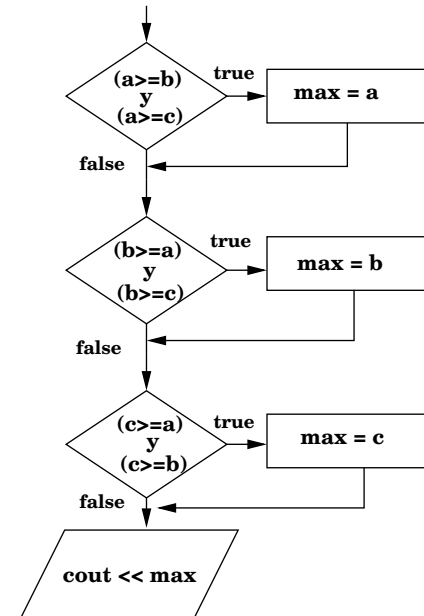
```
    max = b;
```

```
if ((c >= a) && (c >= b))
```

```
    max = c;
```



154



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 evaluaciones de expresiones lógicas.

Ejemplo. El mayor de tres números (segunda aproximación)

Algoritmo: Mayor de tres números. Versión 2

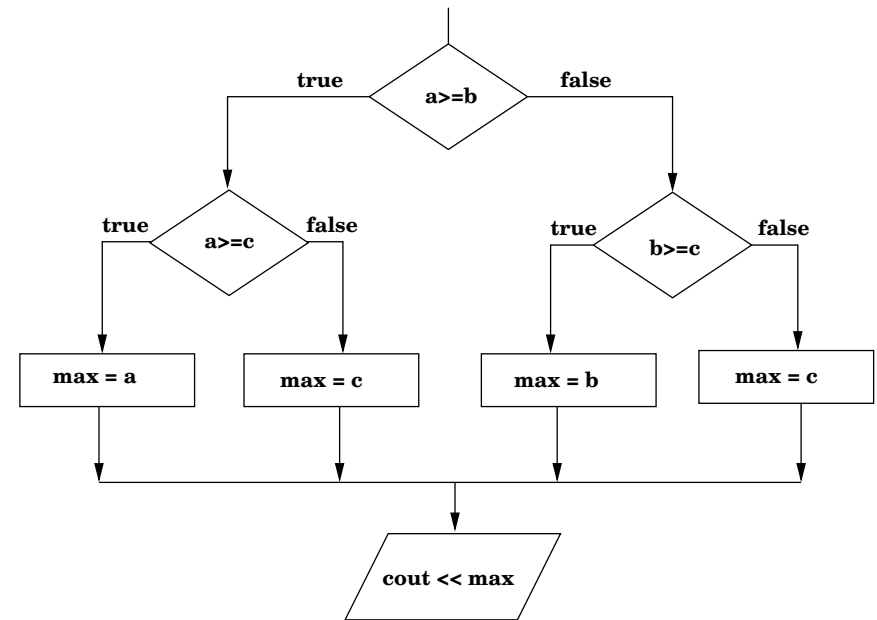
▷ **Entradas y Salidas:** idem

▷ **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;
```



Inconvenientes:

- ▷ Repetimos código: `max = c;`
- ▷ La solución es difícil de extender a más valores.

El mayor de cuatro números con la segunda 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;
```



Ejemplo. El mayor de tres números (tercera aproximación)

Algoritmo: Mayor de tres números. Versión 3

▷ **Entradas y Salidas:** idem

▷ **Descripción e Implementación:**

```
/*
  Calcular el máximo (max) entre a y b.
  Calcular el máximo entre max y c.
*/

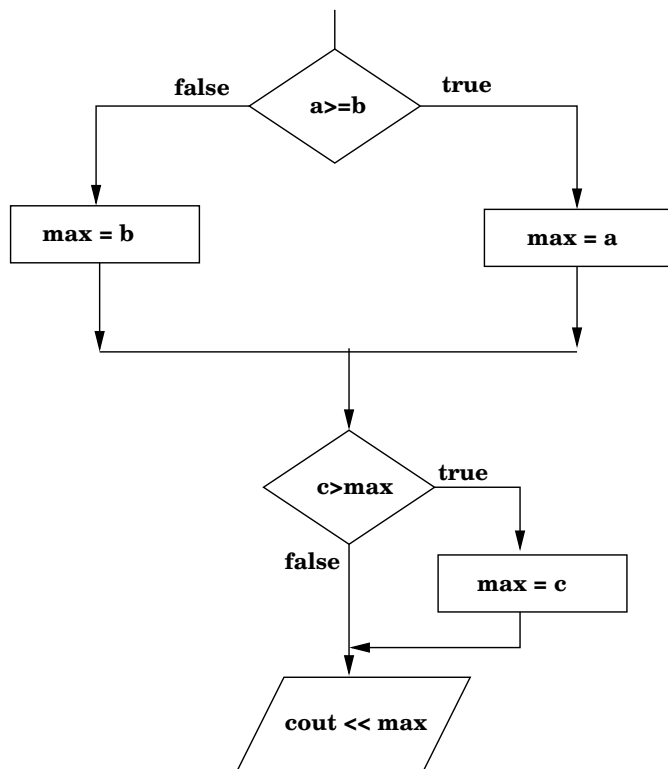
if (a >= b)
    max = a;
else
    max = b;

if (c >= max)
    max = c;
```

Mejora: Con el primer `if` garantizamos que la variable `max` tiene un valor (o bien `a` o bien `b`). Por tanto, en el otro condicional basta usar una desigualdad estricta:

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

if (c > max)
    max = c;
```



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/~carlos/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 dicho valor.

Diseñad los algoritmos para que sean fácilmente extensibles a situaciones más generales.

IMPORTANT

II.1.6.2. 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:

Compruebo 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.
*/
```



▷ **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 (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
```



Usad 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 seguir estas normas baja puntos en el examen.



II.1.6.3. Las expresiones lógicas y el principio de una única vez

Según el principio de una única vez (página 95) no debemos repetir el código de una expresión en distintas partes del programa, si ésta no varía. Lo mismo se aplica si es una expresión lógica.

Ejemplo. Retomamos el ejemplo de subir la nota de la página 131. 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)
    cout << "Examen escrito superado con la nota: " << nota_escrito;
.....
```



Si cambiamos el criterio a que sea mayor estricto, debemos modificar el código de dos líneas. Para resolverlo, 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;
.....
```



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.

Ejemplo. Consideremos la solución del ejercicio de la página 138

```
if (16 <= edad && edad < 18)
    bonificacion = 0.02;
else{
    if (edad <= 35)
        if (meses_trabajados <= 3)
            bonificacion = 0.08;
        else
            bonificacion = 0.04;
    else
        if (meses_trabajados <= 6)
            bonificacion = 0.05;
        else
            bonificacion = 0.03;
}
```



Supongamos que el criterio de experiencia previa considera el mismo número de meses (6) como umbral, independientemente de la edad:

```
if (16 <= edad && edad < 18)
    bonificacion = 0.02;
else{
    if (edad <= 35)
        if (meses_trabajados <= 6)
            bonificacion = 0.08;
        else
            bonificacion = 0.04;
    else
        if (meses_trabajados <= 6)
            bonificacion = 0.05;
        else
            bonificacion = 0.03;
}
```

Si ahora el criterio cambiase a que el número de meses de umbral fuese 4, por ejemplo, habría que modificar dos líneas de código, lo que viola el principio de una única vez. Para resolverlo, introducimos variables lógicas por cada una de las condiciones.

```
bool en_edad_aprendiz;
bool en_edad_joven;
bool es_menor_edad;
bool tiene_poca_experiencia;
.....
es_menor_edad = edad < 18;
en_edad_aprendiz = 16 <= edad && es_menor_edad; // [16,18)
en_edad_joven = !es_menor_edad && edad <= 35; // [18,35]
tiene_poca_experiencia = meses_trabajados <= 6;
```



```
if (en_edad_aprendiz)
    bonificacion = 0.02;
else{
    if (en_edad_joven)
        if (tiene_poca_experiencia)
            bonificacion = 0.08;
        else
            bonificacion = 0.04;
    else
        if (tiene_poca_experiencia)
            bonificacion = 0.05;
        else
            bonificacion = 0.03;
}
```

Y mejor si usamos constantes en vez de literales, según vimos en la página 32:



```
const double BONIF_BAJA      = 0.2;
const double BONIF_MEDIA_BAJA = 0.3;
const double BONIF_MEDIA     = 0.4;
const double BONIF_MEDIA_ALTA = 0.5;
const double BONIF_ALTA      = 0.8;
const int MAYORIA_EDAD       = 18;
const int EDAD_MINIMA_TRABAJAR = 16;
const int LIMITE_JOVEN       = 35;
const int LIMITE_MESES_SIN_EXPERIENCIA = 6;
bool en_edad_aprendiz, en_edad_joven, es_menor_edad;
bool tiene_poca_experiencia;
.....
es_menor_edad = edad < MAYORIA_EDAD;
en_edad_aprendiz = EDAD_MINIMA_TRABAJAR <= edad && es_menor_edad;
en_edad_joven = !es_menor_edad && edad <= LIMITE_JOVEN;
tiene_poca_experiencia = meses_trabajados <= LIMITE_MESES_SIN_EXPERIENCIA;

if (en_edad_aprendiz)
    bonificacion = BONIF_BAJA;
else{
    if (en_edad_joven)
        if (tiene_poca_experiencia)
            bonificacion = BONIF_ALTA;
        else
            bonificacion = BONIF_MEDIA;
    else
        if (tiene_poca_experiencia)
            bonificacion = BONIF_MEDIA_ALTA;
        else
            bonificacion = BONIF_MEDIA_BAJA;
}

cout << "\nBonificación final: " << bonificacion;
```

http://decsai.ugr.es/~carlos/FP/II_bonificacion.cpp

II.1.6.4. Separación de entradas/salidas y cálculos

En temas posteriores se introducirán herramientas para aislar en módulos (clases, funciones, etc) bloques de código. Será importante que los módulos que hagan entradas y salidas de datos no realicen 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 Windows son distintas que en modo consola, por lo que los módulos serán distintos. Pero el módulo de cálculos será el mismo y podrá reutilizarse.

Por ahora, trabajamos en un único fichero y sin módulos. Pero al menos, perseguiremos el objetivo de separar E/S y C, separando los bloques de código de cada parte.

Los bloques que realizan entradas o salidas de datos (cin, cout) estarán separados de los bloques que realizan cálculos.

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.

IMPORTANT

Ejemplo. Retomamos los ejemplos de la página 115. Para no mezclar E/S y C, debemos sustituir el código siguiente:

// Cálculos 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;
.....
// Cálculos: 😊

es_par = entero % 2 == 0;

// Salida de resultados:

if (es_par)
    cout << "\nEs par";
else
    cout << "\nEs impar";

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

De la misma forma, cambiaremos el código siguiente:

```
// Cómputos mezclados con la salida de resultados: 😞  
  
if (dato >= inferior && dato <= superior)  
    cout << "\nEl valor " << dato << " está dentro del intervalo";  
else  
    cout << "\nEl valor " << dato << " está fuera del intervalo";
```

por:

```
bool pertenece_al_intervalo;  
.....  
// Cómputos: 😊  
  
pertenece_al_intervalo = dato >= inferior && dato <= superior;  
  
// Salida de resultados:  
  
if (pertenece_al_intervalo)  
    cout << "\nEl valor " << dato << " está dentro del intervalo";  
else  
    cout << "\nEl valor " << dato << " está fuera del intervalo";
```

¿Y si utilizamos un dato de tipo string?

```
string pertenece_al_intervalo;  
.....  
// Cómputos:  
  
if (dato >= inferior && dato <= superior)  
    pertenece_al_intervalo = "está dentro";  
else  
    pertenece_al_intervalo = "está fuera";  
  
// Salida de resultados:  
  
if (pertenece_al_intervalo == "esta dentro")  
    cout << "\nEl valor " << dato << " está dentro del intervalo";  
else  
    cout << "\nEl valor " << dato << " está fuera del intervalo";
```



¿Localiza el error?

```
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?

Jamás usaremos un tipo string para detectar un número limitado de alternativas posibles ya que es propenso a errores.

Ejemplo. Retomamos el ejemplo de la edad y altura de una persona de la página 135. Para separar E/S y Cómputos, introducimos variables intermedias.

```
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/~carlos/FP/II_altura.cpp

II.1.6.5. 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. Son tokens formados siguiendo las mismas reglas que las usadas para los nombres de los datos. Primero se define el *tipo* (lo haremos antes del `main`) y luego la variable de dicho tipo.

```
#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 CategoricalLaboral
    {administrativo, programador, analista, directivo};

int main(){
    CalificacionECTS nota;
    PuestoPodium podium;
    CategoricalLaboral categoria_laboral;

    nota = CalificacionECTS::A;
    podium = PuestoPodium::primero;
    categoria_laboral = CategoricalLaboral::programador;

    // Las siguientes sentencias dan error de compilación 😊
    // categoria_laboral = CategoricalLaboral::peon
    // categoria_laboral = peon;
    // categoria_laboral = 'a';
    // cin >> categoria_laboral;
    .....
}
```

¿Qué operaciones se pueden hacer sobre un enumerado? Por ahora, sólo tiene sentido la comparación de igualdad.

El tipo enumerado puede verse como una extensión del tipo `bool` ya que nos permite representar más de dos opciones excluyentes.

Ejercicio. Modificad el código del programa que calcula las raíces de una ecuación de segundo grado (página 136) 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";

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;
}
}
```

http://decsai.ugr.es/~carlos/FP/II_ecuacion_segundo_grado.cpp

Ampliación:

Observad 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 caracter_categoria_laboral;
    .....
    cin >> caracter_categoria_laboral;

    if (caracter_categoria_laboral == 'm')
        categoria_laboral = CategoriaLaboral::administrativo
    else if (caracter_categoria_laboral == 'p')
        categoria_laboral = CategoriaLaboral::programador
    else if .....
    .....
    if (caracter_categoria_laboral == CategoriaLaboral::administrativo)
        retencion_fiscal = RETENCION_BAJA;
    else if (caracter_categoria_laboral == CategoriaLaboral::programador)
        retencion_fiscal = RETENCION_MEDIA;

    salario_netto = salario_bruto - salario_bruto * retencion_fiscal/100.0
```

Es cierto que en el código anterior, tenemos que usar literales de carácter concretos, lo cual es propenso a errores. Pero 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.1.6.6. Simplificación de expresiones compuestas: Álgebra de Boole

Debemos intentar simplificar las expresiones lógicas, para que sean fáciles de entender. Usaremos las leyes del Álgebra de Boole, muy similares a las conocidas en Matemáticas elementales como:

$$-(a + b) = -a - b \quad a(b + c) = ab + ac$$

$!(A \mid\mid B)$ equivale a $!A \ \&\& \ !B$
 $!(A \ \&\& \ B)$ equivale a $!A \ \mid\mid \ !B$
 $A \ \&\& \ (B \ \mid\mid \ C)$ equivale a $(A \ \&\& \ B) \ \mid\mid \ (A \ \&\& \ C)$
 $A \ \mid\mid \ (B \ \&\& \ C)$ equivale a $(A \ \mid\mid \ B) \ \&\& \ (A \ \mid\mid \ C)$

Nota. Las dos primeras (relativas a la negación) se conocen como *Leyes de Morgan*. Las dos siguientes son la ley distributiva (de la conjunción con respecto a la disyunción y viceversa).

Ampliación:

Consultad:

http://es.wikipedia.org/wiki/Algebra_booleana
<http://serbal.pntic.mec.es/~cmunoz11/boole.pdf>



Ejemplo. Sobre el ejemplo de la página 163, supongamos que nos hubiesen dicho que una persona es un aprendiz si su edad no está por debajo de 16 (estricto) o por encima de 18:

```
en_edad_aprendiz = !(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)
```

Obviamente, resulta mucho más intuitivo poner:

```
en_edad_aprendiz = (16 <= edad) && (edad < 18);
```

que:

```
en_edad_aprendiz = !(edad < 16 || 18 <= edad);
```

Además, como la comparación con 18 se usaba en varios sitios, introducimos la variable lógica `es_menor_edad`.

```
es_menor_edad = edad < MAYORIA_EDAD;
en_edad_aprendiz = EDAD_MINIMA_TRABAJAR <= edad && es_menor_edad;
```

Fomentad la simplificación de las expresiones lógicas.

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.

```
bool es_contribuyente_especial;
.....
es_contribuyente_especial =
    (16 <= edad && edad <= 65 && ingresos < 7000)
    ||
    (16 <= edad && edad <= 65 && ingresos > 75000);
```

Para simplificar la expresión, sacamos factor común y aplicamos la ley distributiva:

```
bool es_contribuyente_especial;
.....
es_contribuyente_especial =
    (16 <= edad && edad <= 65) && (ingresos < 7000 || ingresos > 75000);
```

De esta forma, no repetimos la expresión `edad >= 16 && edad <= 65`. Y mejoramos el código usando variables lógicas para cada concepto:

```
const int LIMITE_SUP_INGRESOS_BAJOS = 7000;
const int LIMITE_INF_INGRESOS_ALTOS = 75000;
const int EDAD_MINIMA_TRABAJAR = 16;
const int EDAD_MAXIMA_TRABAJAR = 65;
bool es_contribuyente_especial, en_edad_laboral;
bool tiene_ingresos_bajos, tiene_ingresos_altos;
.....
en_edad_laboral =
    EDAD_MINIMA_TRABAJAR <= edad && edad <= EDAD_MAXIMA_TRABAJAR;
tiene_ingresos_bajos = ingresos < LIMITE_SUP_INGRESOS_BAJOS;
tiene_ingresos_altos = ingresos > LIMITE_INF_INGRESOS_ALTOS;
es_contribuyente_especial =
    en_edad_laboral && (tiene_ingresos_bajos || tiene_ingresos_altos)
```



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)

Fomentad 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.1.7. 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%";
```

Recordad 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 consultad:

<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

Evaluación en ciclo corto (Short-circuit evaluation) : El compilador optimiza la evaluación de expresiones lógicas evaluando sus términos de izquierda a derecha hasta que sabe el resultado de la expresión completa (lo que significa que puede dejar términos sin evaluar). 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 ( (a>0) && (b<0) && (c<1) ) ...
```

Supongamos `a = -3`.

- ▷ **Ciclo largo**: El compilador evalúa (innecesariamente) todas las expresiones lógicas
- ▷ **Ciclo corto**: El compilador evalúa sólo la primera expresión lógica.

Por lo tanto, pondremos primero aquellas condiciones que sean más probables de evaluarse como `false`

Nota. Lo mismo pasa cuando el operador es `||`

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 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 de que se use la estructura repetitiva post-test se recomienda usar el siguiente estilo:

```
do{  
    <cuerpo bucle>  
}while (<condición>;
```



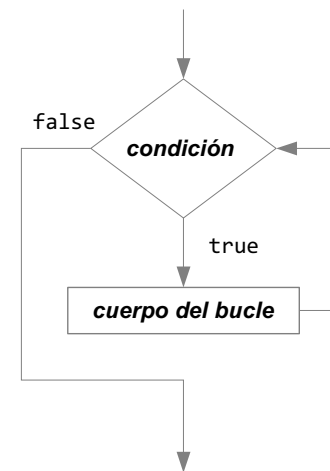
en vez de:

```
do{  
    <cuerpo bucle>  
}  
while (<condición>;
```

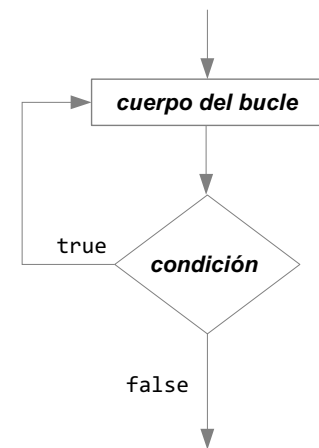


ya que, en la segunda versión, si no vemos las instrucciones antes del while, da la impresión que éste es un pre-test.

Bucle pre-test



Bucle post-test



II.2.1.2. Algunos usos de los bucles

Ejemplo. Crear un *filtro (filter)* de entrada de datos: Leer un valor y no permitir al usuario que lo introduzca fuera de un rango determinado.

```
// Introducir un único valor, pero positivo.
// Imprimir la raíz cuadrada.

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

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:

Observad que el estilo de codificación se rige por las mismas normas que las indicadas en la estructura condicional (página 107)

Ejemplo. Escribir 20 líneas con 5 estrellas cada una.

```
.....
int num_lineas;

num_lineas = 1;

do{
    cout << '\n' << "*****" ;
    num_lineas = num_lineas + 1;    // nuevo = antiguo + 1
}while (num_lineas <= 20);
```

O bien:

```
num_lineas = 0;

do{
    cout << '\n' << "*****" ;
    num_lineas = num_lineas + 1;
}while (num_lineas < 20);
```

O bien:

```
num_lineas = 1;

while (num_lineas <= 20){
    cout << '\n' << "*****" ;
    num_lineas = num_lineas + 1;
}
```

Preferible:

```
num_lineas = 0;    // Llevo 0 líneas impresas

while (num_lineas < 20){
    cout << '\n' << "*****" ;
    num_lineas = num_lineas + 1;
}
```

Dentro del bucle, JUSTO antes de comprobar la condición, la variable total contiene el número de líneas que hay impresas 😊

Nota:

Podríamos usar el operador de incremento:

```
while (num_lineas < 20){
    cout << '\n' << "*****" ;
    num_lineas++;
}
```

Elección entre un bucle pre-test o post-test: ¿Hay situaciones en las que no queremos que se ejecute el cuerpo del bucle?

Sí ⇒ pre-test

No ⇒ post-test

Ejercicio. Leer un número positivo `tope` e imprimir `tope` líneas con 5 estrellas cada una.

```
cout << "\n¿Cuántas líneas de asteriscos quiere imprimir? ";

do{
    cin >> tope;
}while (tope < 0);

num_lineas = 0;

do{
    cout << '\n' << "*****" ;
    num_lineas++;
}while (num_lineas < tope);
```

Problema: ¿Qué ocurre si `tope = 0`?

Ejercicio. Resolved el problema anterior con un bucle pre-test

Consejo: Fomentad 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.



Ejercicio. Calculad 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

```
num_digitos = 1;
```

```
while (n > 9){  
    n = n/10;  
    num_digitos++;  
}
```

Mejor si no modificamos la variable original:

```
n_dividido = abs(n); // valor absoluto  
num_digitos = 1;
```

```
while (n_dividido > 9){  
    n_dividido = n_dividido/10;  
    num_digitos++;  
}
```

http://decsai.ugr.es/~carlos/FP/II_numero_de_digitos.cpp

Ejemplo. Leer un entero tope positivo y escribid los pares \leq tope

```
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 => Imprimelo  
    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 con 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. Realizad 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:



- ▷ 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)
- ▷ Usando variables lógicas.

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/~carlos/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 165. 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.


Solución con una variable lógica:

```
bool seguir_leyendo;

do{
    cin >> numero;

    seguir_leyendo = (numero != TERMINADOR);

    if (seguir_leyendo)
        suma = suma + numero;
}while (seguir_leyendo);
```



// Leemos candidato
// Comprobamos si es el terminador
// Lo procesamos

Esta solución es un poco más ineficiente que la lectura anticipada ya que repetimos la observación de la variable `seguir_leyendo` (en el `if` y en el `while`). En cualquier caso, 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.

Para no repetir código en los bucles que leen datos, o bien usaremos la técnica de lectura anticipada o bien introduciremos variables lógicas que controlen la condición de terminación de lectura.

Ejercicio. Ir leyendo enteros hasta llegar al cero. Imprimir el número de pares e impares leídos.

```
#include <iostream>
using namespace std;
int main(){
    int num_pares, num_impares, valor;

    num_pares = 0;
    num_impares = 0;
    cout << "\nIntroduce valor: ";
    cin >> valor;

    while (          ) {
        if (          )

            ;

        else

            ;

        cout << "\nIntroduce valor: ";
        cin >> valor;
    }

    cout << "\nFueron " << num_pares << " pares y "
          << num_impares << " impares";
}
```

II.2.1.4. Bucles sin fin

Ejemplo. Imprimir los divisores de un valor.

```
int divisor, valor, tope;

cin >> valor;
tope = valor / 2;
divisor = 2;

while (divisor <= tope){
    if (valor % divisor == 0)
        cout << "\n" << divisor << " es un divisor de " << valor;
    divisor++;
}
```

¿Qué pasa si introducimos un else?

```
while (divisor <= tope){
    if (valor % divisor == 0)
        cout << "\n" << divisor << " es un divisor de " << valor;
    else
        divisor++;
}
```

Es un bucle sin fin.

Garantizad que en algún momento, la condición del bucle se hace falsa.

Tened 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. Fomentad el uso de condiciones de desigualdad:

```
contador = 1;

while (contador <= 10) {
    contador = contador + 2;
}
```

II.2.2. Programando como profesionales

II.2.2.1. Condiciones compuestas

Es normal que necesitemos comprobar más de una condición en un bucle. Dependiendo del algoritmo necesitaremos conectarlas con `&&` o con `||`.

Ejemplo. Leer 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:

```
do{
    cout << "Desea formatear el disco";
    cin >> opcion;
    opcion = toupper(opcion);
}while ( opcion != 'S' && opcion != 'N' );
```

Ejemplo. Calcular 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, empezad planteando las condiciones simples que la forman. Pensad cuándo queremos salir del bucle y conectad 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/~carlos/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)
```

Diseñad las condiciones compuestas de forma que sean fáciles de leer

Ejercicio. ¿Qué pasaría si quitásemos el `else`?

II.2.2.2. 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. Comprobar si un número entero positivo es primo. Debemos buscar un divisor suyo. Si lo encontramos, no es primo.

```
int valor, divisor;
bool es_primo;

cout << "Introduzca un numero natural: ";
cin >> valor;

es_primo = true;
divisor = 2
```

```
while (divisor < valor){
    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;
    bool es_primo;

    cout << "Introduzca un numero natural: ";
    cin >> valor;

    es_primo = true;
    divisor = 2;

    while (divisor < valor && 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/~carlos/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.

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;
tope = sqrt(1.0 * valor); // Necesario forzar casting a double
divisor = 2

while (divisor < tope && es_primo){
    if (valor % divisor == 0)
        es_primo = false;
    else
        divisor++;
}
```



II.2.2.3. Evaluación de expresiones dentro y fuera del bucle

En ocasiones, una vez terminado un bucle, debemos 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. Emitid una alarma cuando se encuentren fuera del rango [45, 120] indicando si es baja o alta.

```
int main(){
    const int MIN_LATIDOS = 45;
    const int MAX_LATIDOS = 120;
    int latidos;

    // La siguiente versión repite la evaluación dentro y fuera del bucle
    // de algunas expresiones equivalentes .

    do{
        cin >> latidos;
    }while (MIN_LATIDOS <= latidos && latidos <= MAX_LATIDOS);

    if (latidos < MIN_LATIDOS)
        cout << "\nNúmero de latidos anormalmente bajo: ";
    else
        cout << "\nNúmero de latidos anormalmente alto: ";

    cout << latidos;
}
```



Se repite código ya que `latidos < MIN_LATIDOS` equivale a `!(MIN_LATIDOS <= latidos)` Para resolverlo, introducimos variables lógicas intermedias:

```
int main(){
    const int MIN_LATIDOS = 45;
    const int MAX_LATIDOS = 120;
    int latidos;
    bool frecuencia_cardiaca_baja, frecuencia_cardiaca_alta;

    do{
        cin >> latidos;
        frecuencia_cardiaca_baja = latidos < MIN_LATIDOS;
        frecuencia_cardiaca_alta = latidos > MAX_LATIDOS;
    }while (!frecuencia_cardiaca_baja && !frecuencia_cardiaca_alta);

    if (frecuencia_cardiaca_baja)
        cout << "\nNúmero de latidos anormalmente bajo: ";
    else if (frecuencia_cardiaca_alta) // Realmente no es necesario el if
        cout << "\nNúmero de latidos anormalmente alto: ";

    cout << latidos;
}
```

Y mejor aún, podríamos usar un enumerado en vez de dos bool.

http://decsai.ugr.es/~carlos/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.2.2.4. 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 cuatro. En cualquier caso, 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.

```
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/~carlos/FP/II_notas.cpp

Construid los bucles de forma que no haya que arreglar nada después de su finalización

II.2.2.5. Estilo de codificación

La siguiente implementación del anterior algoritmo es nefasta ya que cuesta mucho trabajo entenderla debido a los identificadores elegidos 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. Hallar la media aritmética de cinco enteros leídos desde teclado.

```
int main(){
    int contador, valor, suma, inicio, final;
    double media;

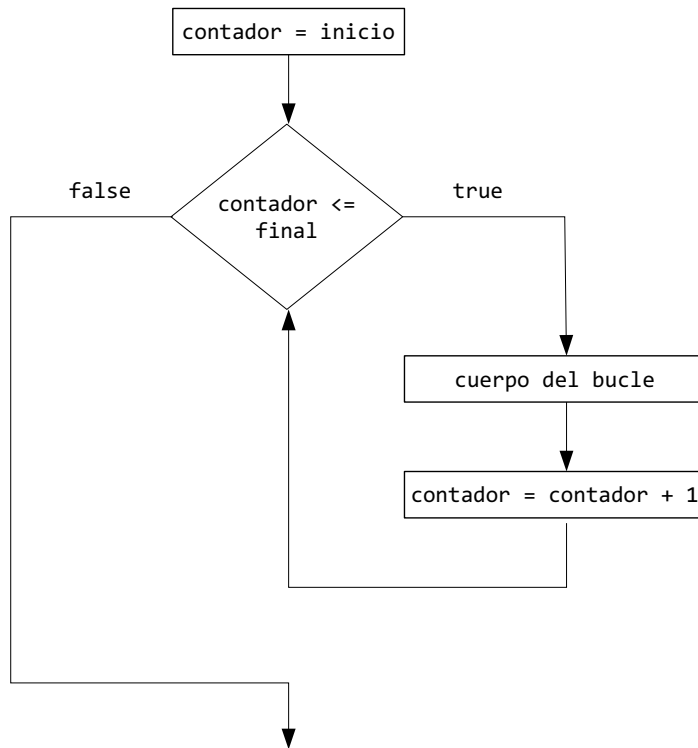
    inicio = 1;
    final = 5;
    suma = 0;
    contador = inicio;

    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:

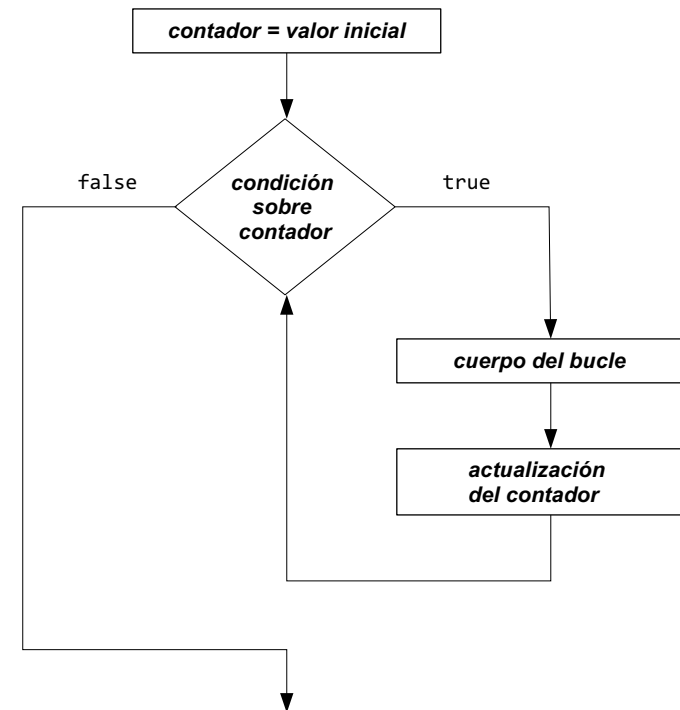


Vamos a implementar el mismo diagrama, pero con otra sintaxis (usando el bucle `for`).

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>
```



Ejemplo. Hallar 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`

Ejemplo. Imprimir los números del 100 al 1. Encuentre errores en este código:

```
For {x = 100, x>=1, x++}  
    cout << x << " ";
```

Ejemplo. Imprimir 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 << " ";  
}
```

Este problema también se podría haber resuelto como sigue:

```
int par;  
  
num_pares = 0;  
  
for (par = -10; par <= 10; par = par + 2)  
    cout << par << " ";
```

Ejemplo. Imprimir 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 \leq final

$\text{final} - \text{inicio} + 1$

▷ Si incremento = 1, $\text{inicio} \leq \text{final}$ y contador $<$ final

$\text{final} - \text{inicio}$

```
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 << "*";
```

Usaremos los bucles for cuando sepamos, antes de entrar al bucle, el número de iteraciones que se tienen que ejecutar.

Nota:

Cuando las variables usadas en los bucles no tienen un significado especial podremos usar nombres cortos como `i`, `j`, `k`

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).



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;
}
```

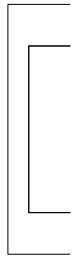
II.2.4. Anidamiento de bucles

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.

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.

Anidamiento correcto



Anidamiento inválido



Ejemplo. Imprimir la tabla de multiplicar de los `TOPE` primeros números.

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

int main(){
    const int TOPE = 3;
    int izda, dcha;

    cout << "Impresión de la tabla de multiplicar del " << TOPE << "\n";

    for (izda = 1 ; izda <= TOPE ; izda++) {
        for (dcha = 1 ; dcha <= TOPE ; dcha++)
            cout << izda << "*" << dcha << "=" << izda * dcha << " ";
        cout << "\n";
    }
}
```

http://decsai.ugr.es/~carlos/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

Observad 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
```

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

Ejemplo. Imprimir en pantalla los primos menores que un entero.

```
int main(){
    int entero, posible_primo, divisor;
    bool es_primo;

    cout << "Introduzca un entero ";
    cin >> entero;
    cout << "\nLos primos menores que " << entero << " son:\n";

    for (posible_primo = entero - 1 ; posible_primo > 1 ; posible_primo--){
        es_primo = true;
        divisor = 2;

        while (divisor < posible_primo && es_primo){
            if (posible_primo % divisor == 0)
                es_primo = false;
            else
                divisor++;
        }

        if (es_primo)
            cout << posible_primo << " ";
    }
}
```

http://decsai.ugr.es/~carlos/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.

Imprimir 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: $n \geq \text{primo}$ 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 207. 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 $n\%\text{primo}$, `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/~carlos/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:

`! dato < 5` \Leftrightarrow `(!dato) < 5` \Leftrightarrow (4 equivale a true) `(!true) < 5` \Leftrightarrow `false < 5` \Leftrightarrow `0 < 5` \Leftrightarrow true

¡Imprime 4 es mayor o igual que 5!

Solución:

```
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 180

```
if (dato >= 5)
    cout << dato << " es mayor o igual que 5";
else
    cout << dato << " es menor de 5";
```

Ejercicio. ¿Qué ocurre en este código y por qué?

```
bool es_menor;

es_menor = 0.2 <= 0.3 <= 0.4;
```

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 da beneficios 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: Evita 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;
```

II.3.2.2. Bucles `for` con sentencias de incremento incorrectas

Lo siguiente es 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` (recordad lo visto en la página 236) 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 (recordar lo visto en la página 223). 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`.

Para mantener dicha finalidad, es necesario respetar la siguiente restricción:

No se debe modificar el valor de la variable controladora, ni el valor final dentro del cuerpo del bucle `for`

IMPORTANT

Sin embargo, C++ no impone dicha restricción. Será responsabilidad del programador.

Ejemplo. Sumar los divisores de `valor`. ¿Dónde está el fallo en el siguiente código?:

```
suma = 0;  
tope = valor/2;  
  
for (divisor = 2; divisor <= tope ; divisor++) {  
    if (valor % divisor == 0)  
        suma = suma + divisor;  
  
    divisor++;  
}
```

El código compila pero se produce un error lógico ya que la variable `divisor` se incrementa en dos sitios.

II.3.2.4. El bucle `for` como ciclo controlado por condición

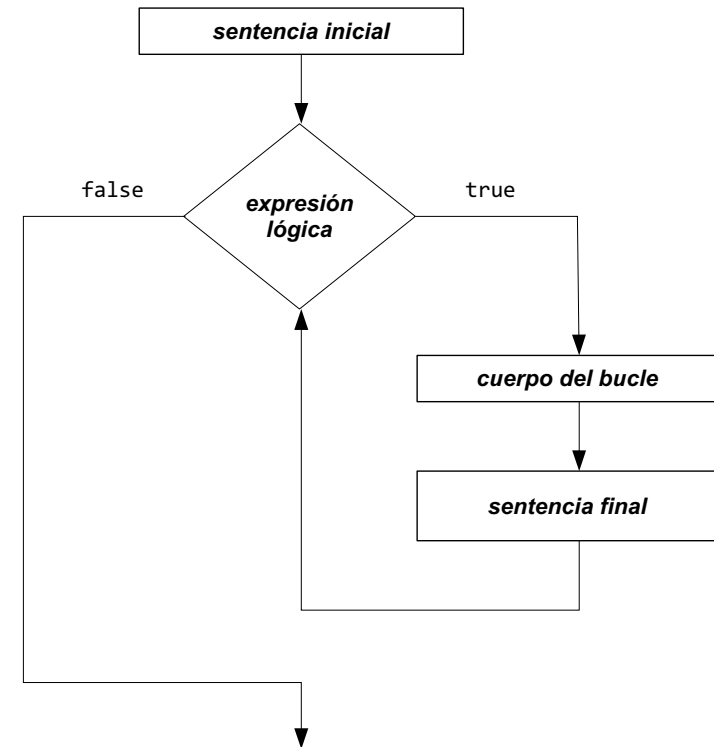
Si bien en muchos lenguajes tales como PASCAL, FORTRAN o BASIC el comportamiento del ciclo `for` es un bucle controlado por contador, en C++ es un ciclo más versátil, controlado por condición.

En el caso concreto de C++, su sintaxis es la siguiente:

```
for ([<sentencia inicial>;  
    [<expresión lógica>;  
    [<sentencia final>])  
    <cuerpo bucle>
```

donde,

- ▷ **<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 arriba para comprobar el valor de la expresión lógica.



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. Escribe los enteros entre `inf` y `sup`, hasta llegar al primer múltiplo de 13:

```
cin >> inf;
cin >> sup;

for (entero = inf; entero <= sup; entero++) {
    cout << entero << " ";

    if (entero % 13 == 0)
        entero = sup + 1;
}
```



El código se ejecuta correctamente pero estamos modificando la variable contadora dentro del bucle, algo que debemos evitar. Podemos usar un `while`:

```
entero = inf;

while (entero <= sup && entero % 13 != 0){
    cout << entero << " ";
    entero++;
}
```



También podemos añadir una condición a la expresión lógica del `for`:

```
for (entero = inf; entero <= sup && entero % 13 != 0 ; entero++)
    cout << entero << " ";
```



Nunca nos saldremos de un bucle `for` asignándole un valor extremo a la variable contadora. Basta con que añadamos una condición a la expresión lógica del `for`

Ejemplo. Comprobar si un número es primo. Lo resolvimos en la página 207:

```
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;
```

Observad 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 (entero = inf; entero<=sup && entero%13 != 0 ; entero++)  
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++.

En resumen, usaremos un bucle `for` en los casos en los que siempre exista:

- ▷ Una sentencia de inicialización del contador
- ▷ Una condición de continuación que involucre al contador (pueden 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 muy recomendable.

Ejemplo. Construir 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: Intentad evitar 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. Sumar 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:

Consultad 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 anteriores, excepto la sentencia `break` con el propósito aquí descrito dentro de la sentencia `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.

Tema III

Funciones y Clases

Objetivos:

- ▷ Introducir el concepto de función y parámetro para no tener que repetir código.
- ▷ Introducir el principio de ocultación de información.
- ▷ Introducir el principio de encapsulación que permitirá empaquetar datos y funciones en un mismo módulo: la Clase.
- ▷ Construir objetos sencillos que cumplan principios básicos de diseño.

Autor: Juan Carlos Cubero.

Sugerencias: por favor, enviar un e-mail a JC.Cubero@decsai.ugr.es

III.1. Funciones

III.1.1. Fundamentos

III.1.1.1. Las funciones realizan una tarea

A la hora de programar, es normal que haya que repetir las mismas acciones con distintos valores. Para no repetir el mismo código, tendremos que:

- ▷ Identificar las acciones repetidas
- ▷ Identificar los valores que pueden variar (esto es, los parámetros)
- ▷ Definir una función que encapsule dichas acciones

Las funciones como `sqrt`, `tolower`, etc., no son sino ejemplos de funciones incluidas en `cmath` y `cctype`, respectivamente, que resuelven tareas concretas y devuelven un valor.

Se suele utilizar una notación prefija, con parámetros (en su caso) separados por comas y encerrados entre paréntesis.

```
int main(){
    double lado1, lado2, hipotenusa, radicando;

    <Asignación de valores a los lados>

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

Si queremos calcular la hipotenusa de dos triángulos rectángulos:

```
int main(){
    double lado1_A, lado2_A, lado1_B, lado2_B,
           hipotenusa_A, hipotenusa_B,
           radicando_A, radicando_B;

    <Asignación de valores a los lados>

    radicando_A = lado1_A*lado1_A + lado2_A*lado2_A;
    hipotenusa_A = sqrt(radicando_A);

    radicando_B = lado1_B*lado1_B + lado2_B*lado2_B;
    hipotenusa_B = sqrt(radicando_B);
    .....
}
```

¿No sería más claro, menos propenso a errores y más reutilizable el código si pudiésemos definir nuestra función `Hipotenusa`? El fragmento de código anterior quedaría como sigue:

```
int main(){
    double lado1_A, lado2_A, lado1_B, lado2_B,
           hipotenusa_A, hipotenusa_B;

    <Asignación de valores a los lados>

    hipotenusa_A = Hipotenusa(lado1_A, lado2_A);
    hipotenusa_B = Hipotenusa(lado1_B, lado2_B);
    .....
}
```

Los lenguajes de programación proporcionan distintos mecanismos para englobar un conjunto de sentencias en un paquete o *módulo* (*module*). En este tema veremos dos tipos de módulos: las funciones y las clases.

III.1.1.2. Definición

```
<tipo> <nombre-función> ([<parám. formales>]) {
    [<sentencias>]

    return <expresión>;
}
```

- ▷ Por ahora, la definición se pondrá después de la inclusión de bibliotecas y antes del `main`. Antes de usar una función en cualquier sitio, hay que poner su definición.
- ▷ Diremos que `<tipo> <nombre-función> (<parám. formales>)` es la *cabecera* (*header*) de la función.
- ▷ El cuerpo de la función debe contener:

```
return <expresión>;
```

donde `<expresión>` ha de ser del mismo tipo que el especificado en la cabecera de la función (también puede ser un tipo *compatible*). El valor que contenga dicha expresión es el valor que devuelve la función cuando es llamada.

```
double Cuadrado(double entrada){
    return entrada*entrada;
}
```

- ▷ La llamada a una función constituye una expresión:
`Cuadrado(valor)` es una expresión.

- ▷ En C++ no se pueden definir funciones dentro de otras. Todas están al mismo nivel.
- ▷ Como estilo de codificación, escribiremos la primera letra de las funciones en mayúscula.

III.1.1.3. Parámetros formales y actuales

- ▷ Los **parámetros formales** (*formal parameters*) son aquellos especificados en la cabecera de la función.
Al declarar un parámetro formal hay que especificar su tipo de dato.
Los parámetros formales sólo se conocen dentro de la función.
- ▷ Los **parámetros actuales** (*actual parameters*) son las expresiones pasadas como argumentos en la llamada a una función. El formato de la llamada es:

`<nombre-función> (<lista parámetros actuales>);`

Nota:

La traducción correcta de *actual* es *real*, pero también es válido el nombre *actual* en español para denotar el valor que se usa en el mismo momento de la llamada.

```
double Cuadrado(double entrada){
    return entrada*entrada;
}

int main(){
    double resultado, valor;

    valor = 4;
    resultado = Cuadrado(valor);    // resultado = 16
    cout << "El cuadrado de " << valor << " es " << resultado;
}
```

► Flujo de control

Cuando se ejecuta la llamada `resultado = Cuadrado(valor);` el flujo de control salta a la definición de la función.

- ▷ Se realiza la correspondencia entre los parámetros.
El correspondiente parámetro formal recibe una copia del parámetro actual, es decir, se realiza la siguiente *asignación en tiempo de ejecución*:

parámetro formal = parámetro actual

En el ejemplo, `entrada = 4`

Esta forma de pasar parámetros se conoce como **paso de parámetro por valor** (*pass-by-value*) .

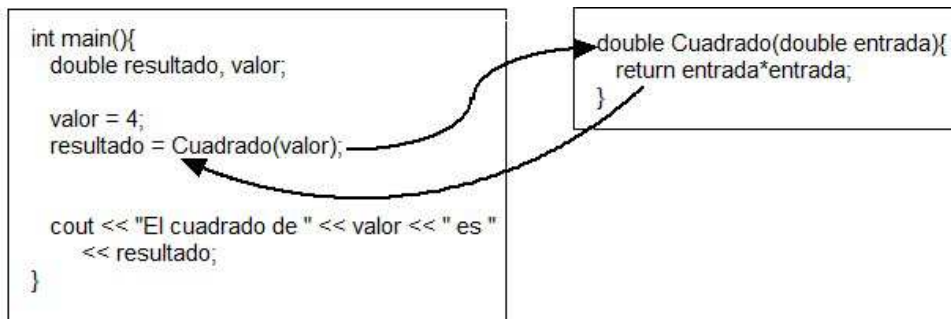
- ▷ Empiezan a ejecutarse las sentencias de la función y, cuando se llega a alguna sentencia `return <expresión>`, termina la ejecución de la función y devuelve el resultado de evaluar `<expresión>` al lugar donde se realizó la invocación.
- ▷ A continuación, el flujo de control prosigue por donde se detuvo al realizar la invocación de la función.

```
double Cuadrado(double entrada){
    return entrada*entrada;
}

int main(){
    double resultado, valor;

    valor = 4;
    resultado = Cuadrado(valor);    // resultado = 16
    cout << "El cuadrado de " << valor << " es "
        << resultado;
}
```

entrada = valor



► Correspondencia entre parámetros actuales y formales

- Debe haber exactamente el *mismo número* de parámetros actuales que de parámetros formales.

```
double Cuadrado(double entrada){
    return entrada*entrada
}

int main(){
    double resultado;
    resultado = Cuadrado(5, 8); // Error en compilación
    .....
```

- Debemos garantizar que el parámetro actual tenga un valor correcto antes de llamar a la función.

```
double Cuadrado(double entrada){
    return entrada*entrada
}

int main(){
    double resultado, valor;
    resultado = Cuadrado(valor); // Error lógico
    .....
```

- La correspondencia se establece por *orden de aparición*, uno a uno y de izquierda a derecha.

```
double Resta(double valor_1, double valor_2){
    return valor_1 - valor_2;
}

int main(){
    double un_valor = 5.0, otro_valor = 4.0;
    double resta;

    resta = Resta(un_valor, otro_valor); // 1.0
    resta = Resta(otro_valor, un_valor); // -1.0
```

- ▷ El parámetro actual puede ser una expresión.

Primero se evalúa la expresión y luego se realiza la llamada a la función.

```
hipotenusa_A = Hipotenusa(3 * lado1_A , 3 * lado2_A);
```

- ▷ Cada parámetro formal y su correspondiente parámetro actual han de ser del *mismo tipo* (o *compatible*)

```
double Cuadrado(double entrada){
    return entrada*entrada
}

int main(){
    double resultado;
    int valor = 7;
    resultado = Cuadrado(valor);    // casting automático
    .....
}
```

Problema: que el tipo del parámetro formal sea más *pequeño* que el actual. Si el parámetro actual tiene un valor que no cabe en el formal, se produce un desbordamiento aritmético, tal y como ocurre en cualquier asignación.

```
int DivisonEntera(int numerador, int denominador){
    return numerador / denominador;
}

int main(){
    ... DivisonEntera(400000000000, 10);    // Desbordamiento
```



- ▷ Dentro de una función, se puede llamar a cualquier otra función que esté definida con anterioridad. El paso de parámetros entre funciones se rige por las mismas normas que hemos visto.

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

double Cuadrado(double entrada){
    return entrada*entrada;
}

double Hipotenusa(double un_lado, double otro_lado){
    return sqrt(Cuadrado(un_lado) + Cuadrado(otro_lado));
}

int main(){
    double lado1_A, lado2_A, lado1_B, lado2_B,
           hipotenusa_A, hipotenusa_B;

    <Asignación de valores a los lados>

    hipotenusa_A = Hipotenusa(lado1_A, lado2_A);
    hipotenusa_B = Hipotenusa(lado1_B, lado2_B);
    .....
}
```

Si cambiamos el orden de definición de las funciones se produce un error de compilación.

Flujo de control:

```
double Hipotenusa(double ladoA, double ladoB){  
    return sqrt(Cuadrado(ladoA) +  
                Cuadrado(ladoB));  
}
```

```
double Cuadrado(double entrada){  
    return entrada*entrada;  
}
```

```
double Hipotenusa(double ladoA, double ladoB){  
    return sqrt(Cuadrado(ladoA) +  
                Cuadrado(ladoB));  
}
```

```
double Cuadrado(double entrada){  
    return entrada*entrada;  
}
```

- ▷ **Las modificaciones del parámetro formal no afectan al parámetro actual. Recordemos que el paso por valor conlleva trabajar con una copia del valor correspondiente al parámetro actual, por lo que éste no se modifica.**

```
double Cuadrado(double entrada){  
    entrada = entrada * entrada;  
    return entrada;  
}  
  
int main(){  
    double resultado, valor = 3;  
  
    resultado = Cuadrado(valor);  
  
    // valor sigue siendo 3  
    .....  
}
```

Ejercicio. Definid la función Hipotenusa

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

double Hipotenusa(double un_lado, double otro_lado){
    return sqrt(un_lado*un_lado + otro_lado*otro_lado);
}

int main(){
    double lado1_A, lado2_A, lado1_B, lado2_B,
           hipotenusa_A, hipotenusa_B;

    <Asignación de valores a los lados>

    hipotenusa_A = Hipotenusa(lado1_A, lado2_A);
    hipotenusa_B = Hipotenusa(lado1_B, lado2_B);
    .....
}
```

http://decsai.ugr.es/~carlos/FP/III_hipotenusa.cpp

Ejercicio. Comprobar si dos reales son iguales, aceptando un margen en la diferencia de 0,000001

```
bool SonIguales (double uno, double otro){
    return (abs(uno - otro) <= 1e-6); // abs = Valor absoluto (cmath)
}
```

Ejercicio. Comprobar si un número es par.

```
bool EsPar(int n){
    if (n % 2 == 0)
        return true;
    else
        return false;
}

int main(){
    int un_numero;
    bool es_par_un_numero;

    cout << "Comprobar si un número es par.\n\n";
    cout << "Introduzca un entero: ";
    cin >> un_numero;

    es_par_un_numero = EsPar(un_numero);

    if (es_par_un_numero)
        cout << un_numero << " es par";
    else
        cout << un_numero << " es impar";
}
```

De forma más compacta:

```
bool EsPar (int n){
    return n % 2 == 0;
}
```

http://decsai.ugr.es/~carlos/FP/III_par.cpp

La siguiente función compila correctamente pero se puede producir un error lógico durante su ejecución:

```
bool EsPar (int n) {  
    if (n%2==0)  
        return true;  
}
```



Una función debería devolver siempre un valor. En caso contrario, el comportamiento es indeterminado.

En resumen:

Definimos una única vez la función y la llamamos donde sea necesario. En la llamada a una función sólo nos preocupamos de saber su nombre y cómo se utiliza (los parámetros y el valor devuelto). Esto hace que el código sea:

▷ **Menos propenso a errores**

Después de un `copy-paste` del código a repetir, si queremos que funcione con otros datos, debemos cambiar una a una todas las apariciones de dichos datos, por lo que podríamos olvidar alguna.

▷ **Más fácil de mantener**

Ante posibles cambios futuros, sólo debemos cambiar el código que hay dentro de la función. El cambio se refleja automáticamente en todos los sitios en los que se realiza una llamada a la función

El programador debe identificar las funciones antes de escribir una sola línea de código. En cualquier caso, no siempre se detectan a priori las funciones, por lo que, una vez escrito el código, si detectamos bloques que se repiten, deberemos englobarlos en una función.

Durante el desarrollo de un proyecto software, primero se diseñan los módulos de la solución y a continuación se procede a implementarlos.

IMPORTANT

III.1.1.4. Ámbito de un dato. Datos locales

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.

- ▷ Dentro de una función podemos declarar constantes y variables. Estas constantes y variables sólo se conocerán dentro de la función, por lo que se les denomina **datos locales (local data)**.
- ▷ De hecho, los parámetros formales se pueden considerar datos locales.

```
<tipo> <nombre-función> (<lista parámetros formales>) {
    [<Constantes Locales>]
    [<Variables Locales>]

    [<Sentencias>]

    return <expresión>;
}
```

Al igual que ocurre con la declaración de variables del `main`, las variables locales a una función no inicializadas a un valor concreto tendrán un valor indeterminado al inicio de la ejecución de la función.

Ejemplo. Calculad el factorial de un valor.

Recordemos la forma de calcular el factorial de un entero n :

```
fact = 1;
for (i = 2; i <= n ; i++)
    fact = fact * i;
```

Definimos la función `Factorial`:

- ▷ La función únicamente necesita conocer el valor de n , que será de tipo `int`.
- ▷ Con un `int` de 32 bits, el máximo factorial computable es 12. Con un `long long` de 64 bits podemos llegar hasta 20.

Así pues, la cabecera será:

```
long long Factorial(int n)
```

- ▷ Para hacer los cálculos del factorial incluimos como datos locales a la función las variable `i` y `fact`. Éstas no se conocen fuera de la función.

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

long long Factorial (int n){
    int i;
    long long fact = 1;

    for (i = 2; i <= n; i++)
        fact = fact * i;

    return fact;
}
```

```
int main(){
    int valor;
    long long resultado;

    cout << "Cálculo del factorial de un número\n";
    cout << "\nIntroduzca un entero: ";
    cin >> valor;

    resultado = Factorial(valor);
    cout << "\nFactorial de " << valor << " = " << resultado;
}
```

http://decsai.ugr.es/~carlos/FP/III_factorial.cpp

Dentro de una función no podemos acceder a los datos locales definidos en otras funciones ni a los de `main`.

```
long long Factorial (int n){
    int i;
    long long fact = 1;

    for (i = 2; i <= valor; i++) // Error de compilación 😊
        fact = fact * i;

    return fact;
}

int main(){
    int valor;
    int resultado;

    cout << i; // Error de compilación 😊
    n = 5; // Error de compilación 😊

    cout << "\nIntroduzca valor";
    cin >> valor;

    resultado = Factorial(valor);
    cout << "\nFactorial de " << valor << " = " << resultado;
}
```

Por tanto, los nombres dados a los parámetros formales pueden coincidir con los nombres de los parámetros actuales: son variables distintas.

```
long long Factorial (int n){      // <- n de Factorial. OK
    int i;
    long long fact = 1;

    for (i = 2; i <= n; i++)
        fact = fact * i;

    return fact;
}

int main(){
    int n = 3;                    // <- n de main. OK
    int resultado;

    resultado = Factorial(n);
    cout << "Factorial de " << n << " = " << resultado;

    // Imprime en pantalla lo siguiente:
    // Factorial de 3 = 6
}
```

Ejemplo. Vimos la función para calcular la hipotenusa de un triángulo rectángulo. Podemos darle el mismo nombre a los parámetros actuales y formales.

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

double Hipotenusa(double un_lado, double otro_lado){
    return sqrt(un_lado*un_lado + otro_lado*otro_lado);
}

int main(){
    double un_lado, otro_lado, hipotenusa;

    cout << "\nIntroduzca primer lado";
    cin >> un_lado;
    cout << "\nIntroduzca segundo lado";
    cin >> otro_lado;

    hipotenusa = Hipotenusa(un_lado, otro_lado);
    cout << "\nLa hipotenusa vale " << hipotenusa;
}
```

Ejercicio. Comprobar si un número es primo (recordad el algoritmo visto en la página 207)

```
bool EsPrimo(int valor){
    bool es_primo;
    int divisor;

    es_primo = true;

    for (divisor = 2 ; divisor < valor && es_primo ; divisor++)
        if (valor % divisor == 0)
            es_primo = false;

    return es_primo;
}

int main(){
    int un_numero;
    bool es_primo;

    cout << "Comprobar si un número es primo.\n\n";
    cout << "Introduzca un entero: ";
    cin >> un_numero;

    es_primo = EsPrimo(un_numero);

    if (es_primo)
        cout << un_numero << " es primo";
    else
        cout << un_numero << " no es primo";
}
```

http://decsai.ugr.es/~carlos/FP/III_primo.cpp

Ejercicio. Calcular el MCD de dos enteros.

```
int MCD(int primero, int segundo){
    /*
     * Vamos dividiendo los dos enteros por todos los
     * enteros menores que el menor de ellos hasta que:
     * - ambos sean divisibles por el mismo valor
     * - o hasta que lleguemos al 1
     */
    bool mcd_encontrado = false;
    int divisor, mcd;

    if (primero == 0 || segundo == 0)
        mcd = 0;
    else{
        if (primero > segundo)
            divisor = segundo;
        else
            divisor = primero;

        mcd_encontrado = false;

        while (!mcd_encontrado){
            if (primero % divisor == 0 && segundo % divisor == 0)
                mcd_encontrado = true;
            else
                divisor--;
        }
        mcd = divisor;
    }

    return mcd;
}
```

```
int main(){
    int un_entero, otro_entero, maximo_comun_divisor;

    cout << "Calcular el MCD de dos enteros.\n\n";
    cout << "Introduzca dos enteros: ";
    cin >> un_entero;
    cin >> otro_entero;

    maximo_comun_divisor = MCD(un_entero, otro_entero);

    cout << "\nEl máximo común divisor de " << un_entero
        << " y " << otro_entero << " es: " << maximo_comun_divisor;
}
```

http://decsai.ugr.es/~carlos/FP/III_mcd_funcion_1.cpp

Ejemplo. Construir una función para leer un entero positivo.

```
int LeePositivo(string mensaje){
    int a_leer;

    cout << mensaje;

    do{
        cin >> a_leer
    }while (a_leer <= 0);

    return a_leer;
}

int main(){
    int salario;

    salario = LeePositivo("\nIntroduzca el salario en miles de euros: ");
    .....
}
```


III.1.1.5. La Pila

Cada vez que se llama a una función, se crea dentro de una zona de memoria llamada *pila (stack)*, un compartimento de trabajo asociado a ella, llamado *marco de pila (stack frame)*.

- ▷ Cada vez que se llama a una función se crea el marco asociado.
- ▷ En el marco asociado a cada función se almacenan, entre otras cosas:
 - Los parámetros formales.
 - Los datos locales (constantes y variables).
 - La *dirección de retorno* de la función.
- ▷ Cuando una función llama a otra, el marco de la función llamada se apila sobre el marco de la función desde donde se hace la llamada (de ahí el nombre de pila). Hasta que no termine de ejecutarse la última función llamada, el control no volverá a la anterior.

Ejemplo. En una competición deportiva, hay cinco equipos y tienen que jugar todos contra todos. ¿Cuántos partidos han de jugarse en total? Se supone que hay una sola vuelta. La respuesta es el número de combinaciones de cinco elementos tomados de dos en dos (sin repetición y no importa el orden)

<http://www.disfrutalasmaticas.com/combinatoria/combinaciones-permutaciones.html>

El combinatorio de dos enteros a y b se define como:

$$\binom{a}{b} = \frac{a!}{b!(a-b)!}$$

Construimos la función Combinatorio que llama a la función Factorial:

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

long long Factorial (int n){
    int i;
    long long fact = 1;

    for (i = 2; i <= n; i++)
        fact = fact * i;

    return fact;
}

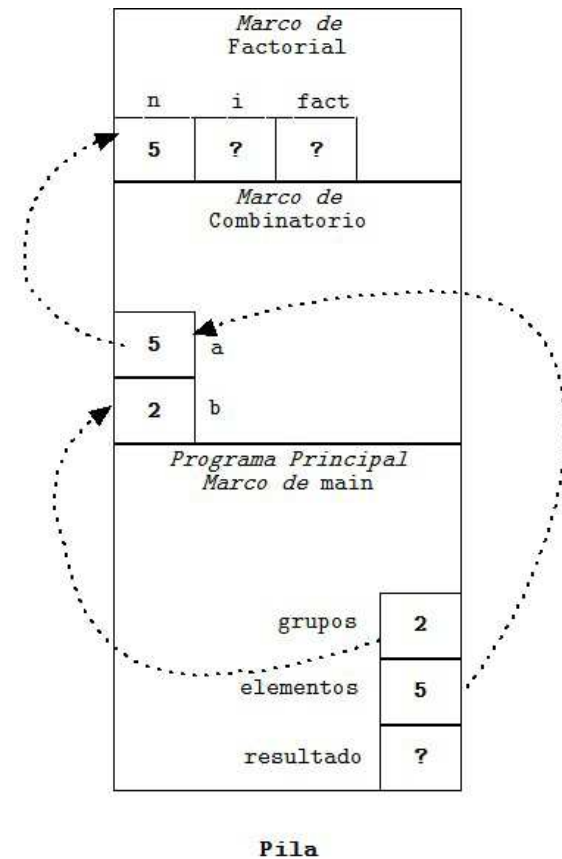
long long Combinatorio(int a, int b){
    return Factorial(a)/(Factorial(b) * Factorial(a-b));
}

int main(){
```

```
int elementos, grupos;
long long resultado;

cout << "Número Combinatorio.\n";
cout << "Introduzca número total de elementos a combinar: ";
cin >> elementos;
cout << "Introduzca cuántos se escogen en cada grupo: ";
cin >> grupos;

resultado = Combinatorio(elementos, grupos);
cout << elementos << " sobre " << grupos << " = " << resultado;
}
```



`main` es de hecho una función como otra cualquiera, por lo que también se almacena en la pila. Es la primera función llamada al ejecutarse el programa.

Ampliación:

La función `main` devuelve un entero al Sistema Operativo y puede tener más parámetros, pero en FP sólo veremos el caso sin parámetros. Por eso, la hemos declarado siempre como:

```
int main(){
    .....
}
```

▷ Si el programa termina con un error, debe devolver un entero distinto de 0.

▷ Si el programa termina sin errores, se debe devolver 0.

Puede indicarse incluyendo `return 0;` al final de `main` (antes de `}`)

En C++, si la función `main` no incluye una sentencia `return` y termina de ejecutarse correctamente el programa, se devuelve 0 por defecto, por lo que podríamos suprimir `return 0;` (sólo en la función `main`)

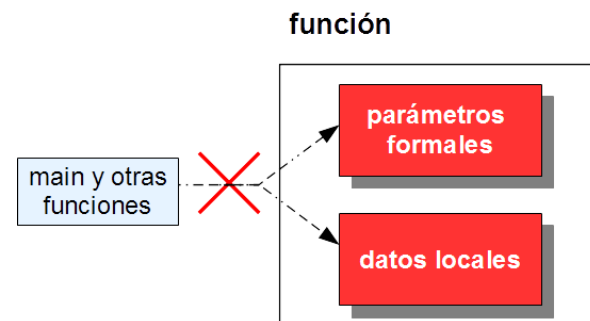


III.1.2. El principio de ocultación de información

¿Qué pasaría si los datos locales fuesen accesibles desde otras funciones?

- ▷ Código propenso a errores, ya que podríamos modificar los datos locales por accidente y provocar errores.
- ▷ Código difícil de mantener, ya que no podríamos cambiar la definición de una función A, suprimiendo, por ejemplo, alguna variable local de A a la que se accediese desde otra función B.

Cada función tiene sus propios datos (parámetros formales y datos locales) y no se conocen en ningún otro sitio. Esto impide que una función pueda interferir en el funcionamiento de otras.



En la llamada a una función no nos preocupamos de saber cómo realiza su tarea. Esto nos permite, por ejemplo, mejorar la implementación de una función sin tener que cambiar una línea de código del programa principal que la llama.

Ejemplo. Sobre el ejercicio que comprueba si un número es primo (página 273) podemos mejorar la implementación, parando al llegar a la raíz cuadrada del valor sin haber encontrado un divisor (ver página 208)

```
bool EsPrimo(int valor){
    bool es_primo;
    int divisor;
    double tope;

    es_primo = true;
    tope = sqrt(valor);

    for (divisor = 2 ; divisor <= tope && es_primo ; divisor++)
        if (valor % divisor == 0)
            es_primo = false;

    return es_primo;
}

int main(){
    .....
    No cambia nada
    .....
}
```

No importa si el lector no comprende el mecanismo matemático anterior.

Al haber ocultado información (los datos locales no se conocen fuera de la función), los cambios realizados dentro de la función no afectan al exterior. Ésto me permite seguir realizando las llamadas a la función al igual que antes (la cabecera no ha cambiado).

Ejemplo. Retomamos el ejemplo del número combinatorio de la página 278. Para calcular los resultados posibles de la lotería primitiva, habría que calcular el número combinatorio 45 sobre 6, es decir, $\text{elementos} = 45$ y $\text{grupos} = 6$. El resultado es 13.983.816, que cabe en un long long. Sin embargo, el cómputo del factorial se desborda a partir de 20.

Para resolver este problema, la implementación de la función Combinatorio puede mejorarse simplificando la expresión matemática:

$$\frac{a!}{b!(a-b)!} = \frac{a(a-1) \cdots (a-b+1)(a-b)!}{b!(a-b)!} = \frac{a(a-1) \cdots (a-b+1)}{b!}$$

Cambiamos la implementación de la función como sigue:

```
long long Combinatorio(int a, int b){
    int numerador, fact_b;

    numerador = 1;
    fact_b = 1;

    for (int i = 1 ; i <= b ; i++){
        fact_b = fact_b*i;
        numerador = numerador * (a - i + 1);
    }

    return numerador/fact_b;
}

int main(){
    No cambia nada
}
```

http://decsai.ugr.es/~carlos/FP/III_combinatorio.cpp

Al estar aislado el código de una función dentro de ella, podemos cambiar la implementación interna de la función sin que afecte al resto de funciones (siempre que se mantenga inalterable la cabecera de la misma)

Lo visto anteriormente puede generalizarse en el siguiente principio básico en Programación:

Principio de Programación:

Ocultación de información (Information Hiding)

Al usar un componente software, no deberíamos tener que preocuparnos de sus detalles de implementación.



Como caso particular de componente software tenemos las funciones y los datos locales a ellas nos permiten ocultar los detalles de implementación de una función al resto de funciones.

III.1.3. Funciones void

Ejemplo. Queremos construir un programa para calcular la hipotenusa de un triángulo rectángulo con una presentación al principio de la siguiente forma:

```
int i, tope_lineas;

.....

for (i=1; i<=tope_lineas ; i++)
    cout << "\n*****";
cout << "Programa básico de Trigonometría";
for (i=1; i<=tope_lineas ; i++)
    cout << "\n*****";

.....
```

¿No sería más fácil de entender si el código del programa principal hubiese sido el siguiente?

```
.....
Presentacion(tope_lineas);
.....
```

En este ejemplo, `Presentacion` resuelve la tarea de realizar la presentación del programa por pantalla, pero no calcula (devuelve) ningún valor, como sí ocurre con las funciones `sqrt` o `Hipotenusa`. Por eso, su llamada constituye una sentencia y no aparece dentro de una expresión. Este tipo particular de funciones que no devuelven ningún valor, se definen como sigue:

```
void <nombre-función> (<lista parámetros formales>) {  
    [<Constantes Locales>]  
    [<Variables Locales>]  
    [<Sentencias>]  
}
```

Obsérvese que no hay sentencia `return`. La función `void` termina cuando se ejecuta la última sentencia de la función.

El paso de parámetros y la definición de datos locales sigue las mismas normas que el resto de funciones.

Para llamar a una función `void`, simplemente, ponemos su nombre y la lista de parámetros actuales con los que realizamos la llamada:

```
void MiFuncionVoid(int parametro_formal, double otro_parametro_formal){  
    .....  
}  
  
int main(){  
    int parametro_actual;  
    double otro_parametro_actual;  
    .....  
    MiFuncionVoid(parametro_actual, otro_parametro_actual);  
    .....
```

Nota:

Los lenguajes suelen referirse a este tipo de funciones con el nombre [procedimiento](#) (*procedure*)

```
#include <iostream>  
#include <cmath>  
using namespace std;  
  
double Cuadrado(double entrada){  
    return entrada*entrada;  
}  
  
double Hipotenusa(double un_lado, double otro_lado){  
    return sqrt(Cuadrado(un_lado) + Cuadrado(otro_lado));  
}  
  
void Presentacion(int tope_lineas){  
    int i;  
    for (i = 1; i <= tope_lineas ; i++)  
        cout << "\n*****";  
    cout << "Programa básico de Trigonometría";  
    for (i = 1; i <= tope_lineas ; i++)  
        cout << "\n*****";  
}  
  
int main(){  
    double lado1, lado2, hipotenusa;  
  
    Presentacion(3);  
  
    cout << "\n\nIntroduzca los lados del triángulo rectángulo: ";  
    cin >> lado1;  
    cin >> lado2;  
  
    hipotenusa = Hipotenusa(lado1,lado2);  
  
    cout << "\nLa hipotenusa vale " << hipotenusa;  
}
```

Como cualquier otra función, las funciones `void` pueden llamarse desde cualquier otra función (`void` o cualquier otra), siempre que su definición vaya antes.

Ejercicio. Aislad la impresión de las líneas de asteriscos en una función aparte, por si quisiéramos usarla en otros sitios.

```
void ImprimeLineas (int num_lineas){
    int i;
    for (i = 1; i <= num_lineas ; i++)
        cout << "\n*****";
}

void Presentacion(int tope_lineas){
    ImprimeLineas (tope_lineas);
    cout << "Programa básico de Trigonometría";
    ImprimeLineas (tope_lineas);
}
```

III.1.4. Ámbito de un dato (revisión)

Recordad que el ámbito de un dato es el conjunto de todos aquellos sitios que pueden acceder a él.

► **Lo que ya sabemos:**

Datos locales (declarados dentro de una función)

- ▷ El ámbito es la propia función.
- ▷ No se puede acceder al dato desde otras funciones.

Nota. Lo anterior se aplica también, como caso particular, a la función `main`

Parámetros formales de una función

- ▷ Se consideran datos locales de la función.

► **Lo nuevo (ni bueno ni malo):**

Datos declarados dentro de un bloque

- ▷ El ámbito es el propio bloque. El ámbito de la variable termina con la llave } que cierra la estructura condicional, repetitiva, etc.

- En un bucle controlado por condición, la variable pierde el valor antiguo en cada iteración:

```
while (condicion){  
    int i;  
    i = 0;  
    .....  
}
```

Cada vez que entra en el bucle se asigna `i = 0`.

- En un bucle `for`, la variable conserva el valor de la iteración anterior:

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

Sólo se asigna `i = 0` en la primera iteración.


Cuando necesitemos puntualmente variables intermedias para realizar nuestros cálculos, será útil definirlos en el ámbito del bloque de instrucciones correspondiente.

Pero tened 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.

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 Suma_desde_0_hasta(int tope){  
    int suma = 0;                // <- suma (ámbito: función)  
  
    while (tope>0){  
        int suma = 0;            // <- suma (ámbito: bloque)  
        suma = suma + tope;      // <- suma (ámbito: bloque)  
        tope--;  
    }  
  
    return suma;                 // <- suma (ámbito: función)  
}  
  
int main(){  
    int tope = 5, resultado;  
  
    resultado = Suma_desde_0_hasta(tope);  
    cout << "Suma hasta " << tope << " = " << resultado;  
  
    // Imprime en pantalla lo siguiente:   
    // Suma hasta 5 = 0  
}
```


► **Lo nuevo (muy malo):**

Datos globales (global data)

- ▷ Son datos declarados fuera de las funciones y del main.
- ▷ El ámbito de los datos globales está formado por todas las funciones que hay definidas con posterioridad 😞

El uso de variables globales permite que todas las funciones las puedan modificar. Esto es pernicioso para la programación, fomentando la aparición de *efectos laterales (side effects)*.

Ejemplo. Supongamos un programa para la gestión de un aeropuerto. Tendrá dos funciones: `GestionMaletas` y `ControlVuelos`. El primero controla las cintas transportadoras y como máximo puede gestionar 50 maletas. El segundo controla los vuelos en el área del aeropuerto y como máximo puede gestionar 30. El problema es que ambos van a usar el mismo dato global `Max` para representar dichos máximos.

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

int Max;           // Variables globales 😞
bool saturacion;

void GestionMaletas(){
    Max = 50;

    if (NumMaletasActual <= Max)
        [acciones maletas]
    else
        ActivarEmergenciaMaletas();
}

void ControlVuelos(){
    if (NumVuelosActual <= Max)
        [acciones vuelos]
    else{
        ActivarEmergenciaVuelos();
        saturacion = true;
    }
}

int main(){
    Max = 30;
    saturacion = false;

    while (!saturacion){
        GestionMaletas(); // Efecto lateral: Max = 50
        ControlVuelos();
    }
    .....
}
```



El uso de variables globales hace que los programas sean mucho más difíciles de depurar ya que la modificación de éstas puede hacerse en cualquier función del programa.

El que un lenguaje como C++ permita asignar un ámbito global a una variable, no significa que esto sea adecuado o recomendable.

El uso de variables globales puede provocar graves efectos laterales, por lo que su uso está completamente prohibido en esta asignatura.



Nota. El uso de *constantes globales* es menos perjudicial ya que, si no pueden modificarse, no se producen efectos laterales

III.1.5. Parametrización de funciones

Los parámetros nos permiten ejecutar el código de las funciones con distintos valores:

```
bool es_par;  
int entero, factorial;  
.....  
es_par    = EsPar(3);  
es_par    = EsPar(entero);  
factorial = Factorial(5);  
factorial = Factorial(4);  
factorial = Factorial(entero);
```

En estos ejemplos era evidente los parámetros que teníamos que pasar. En casos más complejos, no será así.

A la hora de establecer cuáles han de ser los parámetros de una función, debemos determinar:

- ▷ Qué puede cambiar de una llamada a otra de la función.
- ▷ Qué factores influyen en la tarea que la función resuelve.

Ejemplo. Retomemos el ejemplo `ImprimeLineas` de la página 289.

```
void ImprimeLineas (int num_lineas){
    for (int i = 1; i <= num_lineas ; i++)
        cout << "\n*****";
}

void Presentacion(int tope_lineas){
    ImprimeLineas (tope_lineas);
    cout << "Programa básico de Trigonometría";
    ImprimeLineas (tope_lineas);
}
```

`ImprimeLineas` siempre imprime 12 asteriscos. ¿Y si también queremos que sea variable el número de asteriscos? Basta añadir un parámetro:

```
void ImprimeLineas (int num_lineas, int numero_asteriscos){
    for (int i = 1; i <= num_lineas ; i++)
        cout << "\n";

    for (int j = 1; j <= numero_asteriscos; j++)
        cout << "*";
}
```

Ahora podemos llamar a `ImprimeLineas` con cualquier número de asteriscos. Por ejemplo, desde `Presentacion`:

```
void Presentacion(int tope_lineas){
    ImprimeLineas (tope_lineas, 12);
    cout << "Programa básico de Trigonometría";
    ImprimeLineas (tope_lineas, 12);
}
```

En este caso, `Presentacion` siempre imprimirá líneas con 12 asteriscos. Si también queremos permitir que cambie este valor, basta con incluirlo como parámetro:

```
void Presentacion(int tope_lineas, int numero_asteriscos){
    ImprimeLineas (tope_lineas, numero_asteriscos);
    cout << "Programa básico de Trigonometría";
    ImprimeLineas (tope_lineas, numero_asteriscos);
}
```

Es posible que deseemos aislar la impresión de una línea de asteriscos en una función, para así poder reutilizarla en otros contextos. Nos quedaría:

```
void ImprimeAsteriscos (int num_asteriscos){
    for (int i = 1 ; i <= num_asteriscos ; i++)
        cout << "*";
}

void ImprimeLineas (int num_lineas, int num_asteriscos){
    for (int i = 1; i <= num_lineas ; i++){
        cout << "\n";
        ImprimeAsteriscos(num_asteriscos);
    }
}

void Presentacion(int tope_lineas, int num_asteriscos){
    ImprimeLineas (tope_lineas, num_asteriscos);
    cout << "Programa básico de Trigonometría";
    ImprimeLineas (tope_lineas, num_asteriscos);
}
```

Finalmente, si queremos poder cambiar el título del mensaje, basta pasarlo como parámetro:

```
void Presentacion(string mensaje, int tope_lineas, int num_asteriscos){
    ImprimeLineas (tope_lineas, num_asteriscos);
    cout << mensaje;
    ImprimeLineas (tope_lineas, num_asteriscos);
}
```

El programa principal quedaría así:

```
int main(){
    double lado1, lado2, hipotenusa;

    Presentacion("Programa básico de Trigonometría", 3, 32);

    cout << "\n\nIntroduzca los lados del triángulo rectángulo: ";
    cin >> lado1;
    cin >> lado2;

    hipotenusa = Hipotenusa(lado1,lado2);

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

http://decsai.ugr.es/~carlos/FP/III_hipotenusa_presentacion.cpp

Los parámetros nos permiten aumentar la flexibilidad en el uso de la función.

Ejemplo. Retomemos el ejemplo de la altura del tema II. Si el criterio es el de la página 127, una persona es alta si mide más de 190 cm:

```
bool EsAlta (int altura){
    return altura >= 190;
}
```

Pero si el criterio es el de la página 135, para determinar si una persona es alta también influye la edad, por lo que debemos incluirla como parámetro:

```
bool EsMayorEdad (int edad){
    return edad >= 18;
}
```

```
bool EsAlta (int altura, int edad){
    if (EsMayorEdad(edad))
        return altura >= 190;
    else
        return altura >= 175;
}
```

```
int main(){
    int edad, altura;
    bool es_alta, es_mayor_edad;

    cout << "Determinar si una persona es mayor de edad y si es alta.\n\n";
    cout << "Introduzca los valores de edad y altura: ";
    cin >> edad;
    cin >> altura;

    es_mayor_edad = EsMayorEdad(edad);
    es_alta = EsAlta(altura, edad);
}
```

```
if (es_mayor_edad)
    cout << "\nEs mayor de edad";
else
    cout << "\nEs menor de edad";

if (es_alta)
    cout << "\nEs una persona alta";
else
    cout << "\nNo es una persona alta";
}
```

Al diseñar la cabecera de una función, el programador debe analizar detalladamente cuáles son los factores que influyen en la tarea que ésta resuelve.

Si la función es muy simple, podemos usar los literales 190, 175. Pero si es más compleja, seguimos el mismo consejo que vimos en el tema I y usamos constantes tal y como hicimos en la página 170:

```
bool EsMayorEdad (int edad){
    const int MAYORIA_EDAD = 18;

    return edad >= MAYORIA_EDAD;
}

bool EsAlta (int altura, int edad){
    const int UMBRAL_ALTURA_JOVENES = 175,
            UMBRAL_ALTURA_ADULTOS = 190;
    int umbral_altura;

    if (EsMayorEdad(edad))
        umbral_altura = UMBRAL_ALTURA_ADULTOS;
    else
        umbral_altura = UMBRAL_ALTURA_JOVENES;

    return altura >= umbral_altura;
}

int main(){
    <No cambia nada>
}
```

http://decsai.ugr.es/~carlos/FP/III_altura.cpp

Nota:

Si las constantes se utilizasen en otros sitios del programa, podrían ponerse como constantes globales

III.1.6. Programando como profesionales

III.1.6.1. Cuestión de estilo

¿Podemos incluir una sentencia `return` en cualquier sitio de la función?. Poder, se puede, pero no es una buena idea.

Ejemplo. ¿Qué ocurre en el siguiente código?

```
long long Factorial (int n) {  
    int i;  
    long long fact;  
  
    fact = 1;  
    for (i = 2; i <= n; i++) {  
        fact = fact * i;  
  
        return fact;  
    }  
}
```



Si $n \geq 2$ se entra en el bucle, pero la función termina cuando se ejecuta la primera iteración, por lo que devuelve 2. Y si $n < 2$, no se ejecuta ningún `return` y por tanto el comportamiento es indeterminado.

Ejemplo. Considerad la siguiente implementación de la función `EsPrimo`:

```
bool EsPrimo(int valor){  
    int divisor;  
  
    for (divisor = 2 ; divisor < valor; divisor++)  
        if (valor % divisor == 0)  
            return false;  
  
    return true;  
}
```



Sólo ejecuta `return true` cuando no entra en el condicional del bucle, es decir, cuando no encuentra ningún divisor (y por tanto el número es primo) Por lo tanto, la función se ejecuta correctamente pero el código es difícil de entender.

En vez de incluir un `return` dentro del bucle para salirnos de él, usamos, como ya habíamos hecho en este ejemplo, una variable `bool`. Ahora, viendo la cabecera del bucle, vemos todas las condiciones que controlan el bucle:

```
bool EsPrimo(int valor){  
    int divisor;  
    bool es_primo;  
  
    es_primo = true;  
  
    for (divisor = 2 ; divisor < valor && es_primo; divisor++)  
        if (valor % divisor == 0)  
            es_primo = false;  
  
    return es_primo;  
}
```



Así pues, debemos evitar construir funciones con varias sentencias `return` *perdidas* dentro del código. En el caso de funciones con muy pocas líneas de código y siempre que el código quede claro, sí podríamos aceptarlo:

```
bool EsPar (int n){
    if (n % 2 == 0)
        return true;
    else
        return false;
}
```

Consejo: Salvo en el caso de funciones con muy pocas líneas de código, evitad la inclusión de sentencias `return` *perdidas* dentro del código de la función. Usad mejor un único `return` al final de la función.



III.1.6.2. Diseño de la cabecera de una función

¿Qué es mejor: muchos parámetros o pocos? Los justos y necesarios.

Ejemplo. Demasiados parámetros:

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

long long Factorial (int i, int n){
    long long fact = 1;

    for (i = 2; i <= n; i++)
        fact = fact * i;

    return fact;
}

int main(){
    int n = 3;
    long long resultado;
    int que_pinta_esta_variable_en_el_main = 1;

    resultado = Factorial(que_pinta_esta_variable_en_el_main, n);
    cout << "Factorial del número: " << resultado;
}
```



Todos los datos auxiliares que la función necesite para realizar sus cálculos serán datos locales.

Ejemplo. Faltan parámetros:

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

long long Factorial(){
    long long fact = 1;
    int n;

    cin >> n;

    for (i = 2; i <= n; i++)
        fact = fact * i;

    return fact;
}

int main(){
    int resultado;

    resultado = Factorial();
    cout << "Factorial de " << n << " = " << resultado;
}
```



Al leer el valor de `n` dentro de la función, ya no podemos usarla en otras plataformas y tampoco podemos usarla con enteros arbitrarios (que no provengan de la entrada estándar)

Las funciones que realicen un cómputo, no harán también operaciones de E/S



Ejemplo. Demasiados parámetros:

Re-escribimos el ejemplo del MCD de la página 274. La función busca un divisor de dos números, empezando por el menor de ellos. ¿Y si la función exige que se calcule dicho mínimo fuera?

```
int MCD(int primero, int segundo, int el_menor_entre_ambos){
    bool mcd_encontrado = false;
    int divisor, mcd;

    if (primero == 0 || segundo == 0)
        mcd = 0;
    else{
        divisor = el_menor_entre_ambos;
        mcd_encontrado = false;

        while (!mcd_encontrado){
            if (primero % divisor == 0 && segundo % divisor == 0)
                mcd_encontrado = true;
            else
                divisor--;
        }
        mcd = divisor;
    }

    return mcd;
}

int main(){
    int un_entero, otro_entero, menor, maximo_comun_divisor;

    cout << "Calcular el MCD de dos enteros.\n\n";
    cout << "Introduzca dos enteros: ";
    cin >> un_entero;
```




```
cin >> otro_entero;

if (un_entero < otro_entero)
    menor = un_entero;
else
    menor = otro_entero;

maximo_comun_divisor = MCD(un_entero, otro_entero, menor);

cout << "\nEl máximo común divisor de " << un_entero
    << " y " << otro_entero << " es: " << maximo_comun_divisor;
}
```



Para que la nueva versión de la función `MCD` funcione correctamente, **siempre** debemos calcular el menor, antes de llamar a la función. Si no lo hacemos bien, la función no realizará correctamente sus cálculos.

Observad que el parámetro `el_menor_entre_ambos` depende de los valores de los otros dos parámetros. Debemos evitar este tipo de dependencias que normalmente indican la existencia de otra función que los relaciona.

Por tanto, nos quedamos con la solución de la página 274, en la que se calcula el menor dentro de la función.

- ▷ *Si el correcto funcionamiento de una función depende de la realización previa de una serie de instrucciones, éstas deben ir dentro de la función.*
- ▷ *Los parámetros actuales deben poder variar de forma independiente unos de otros.*

Si se prevé su reutilización en otros sitios, el cómputo del menor puede definirse en una función aparte:

```
int Minimo(int un_entero, int otro_entero){
    if (un_entero < otro_entero)
        return un_entero;
    else
        return otro_entero;
}

int MCD(int primero, int segundo){
    bool mcd_encontrado = false;
    int divisor, mcd;

    if (primero == 0 || segundo == 0)
        mcd = 0;
    else{
        divisor = Minimo(primero, segundo);
        mcd_encontrado = false;

        while (!mcd_encontrado){
            if (primero % divisor == 0 && segundo % divisor == 0)
                mcd_encontrado = true;
            else
                divisor--;
        }
        mcd = divisor;
    }

    return mcd;
}

int main(){
```



```
int un_entero, otro_entero, maximo_comun_divisor;

cout << "Calcular el MCD de dos enteros.\n\n";
cout << "Introduzca dos enteros: ";
cin >> un_entero;
cin >> otro_entero;

maximo_comun_divisor = MCD(un_entero, otro_entero);

cout << "\nEl máximo común divisor de " << un_entero
    << " y " << otro_entero << " es: " << maximo_comun_divisor;
}
```

http://decsai.ugr.es/~carlos/FP/III_mcd_funcion_2.cpp

III.1.6.3. Precondiciones

Una **precondición** (*precondition*) de una función es toda aquella restricción que deben satisfacer los parámetros para que la función pueda ejecutarse sin problemas.

```
long long Factorial (int n){
    int i;
    long long fact = 1;

    for (i = 2; i <= n; i++)
        fact = fact * i;

    return fact;
}
```

Si pasamos como parámetro a n un valor mayor de 20, se produce un desbordamiento aritmético. Indicamos esta precondición como un comentario antes de la función.

```
// Prec: 0 <= n <= 20
long long Factorial (int n){
    .....
}

// Prec: 0 <= a, b <= 20
long long Combinatorio(int a, int b){
    .....
}
```

¿Debemos comprobar dentro de la función si se satisfacen sus precondiciones? Y si lo hacemos, ¿qué acción debemos realizar en caso de que no se cumplan? Contestaremos a estas preguntas en la página 403

III.1.6.4. Documentación de una función

Hay dos tipos de comentarios:

► Descripción del algoritmo que implementa la función

- ▷ Describen **cómo** se resuelve la tarea encomendada a la función.
- ▷ Se incluyen **dentro** del código de la función
- ▷ Sólo describimos la esencia del algoritmo (tema II)

Ejemplo. El mayor de tres números

```
int Max3 (int a, int b, int c){  
    int max;  
    /* Calcular el máximo entre a y b -> max  
       Calcular el máximo entre max y c      */  
  
    if (a > b)  
        max = a;  
    else  
        max = b;  
  
    if (c > max)  
        max = c;  
  
    return max;  
}
```

En el examen es imperativo incluir la descripción del algoritmo.

IMPORTANT

► Descripción de la cabecera de la función

- ▷ Describen **qué** tarea resuelve la función.
- ▷ También describen los parámetros (cuando no sea obvio).
- ▷ Se incluyen **fuera**, justo antes de la cabecera de la función

```
// Calcula el máximo de tres enteros
```

```
int Max3 (int a, int b, int c){  
    int max;  
    /* Calcular el máximo entre a y b  
       Calcular el máximo entre max y c  
    */  
  
    if (a > b)  
        max = a;  
    else  
        max = b;  
  
    if (c > max)  
        max = c;  
  
    return max;  
}
```

Incluiremos:

- ▷ Una descripción breve del cometido de la función.

Consejo: *Si no podemos resumir el cometido de una función en un par de líneas a lo sumo, entonces la función es demasiado compleja y posiblemente debería dividirse en varias funciones.*



- ▷ Una descripción de lo que representan los parámetros (salvo que el significado sea obvio) También incluiremos las precondiciones de la función.

```
// Combinatorio de dos números
// Prec: 0 <= a, b <= 20
long long Combinatorio(int a, int b){
    return Factorial(a)/(Factorial(b) * Factorial(a-b));
}
```

Ampliación:

Consultad el capítulo 19 del libro Code Complete de Steve McConnell sobre normas para escribir comentarios claros y fáciles de mantener.



III.2. Clases

Hemos visto cómo las funciones introducen mecanismos para cumplir dos principios básicos:

- ▷ Principio de una única vez.
Identificando tareas y empaquetando las instrucciones que la resuelven.
- ▷ Principio de ocultación de información.
A través de los datos locales.

En general, los lenguajes de programación proporcionan mecanismos de **modularización (modularization)** de código, para así poder cumplir dichos principios básicos.

En la siguiente sección introducimos las **clases**. Proporcionan un mecanismo complementario que permite cumplir los principios de programación:

- ▷ Principio de una única vez.
Identificando objetos y encapsulando en ellos **datos** y **funciones**.
- ▷ Principio de ocultación de información.
Definiendo nuevas reglas de ámbito para los datos.

III.2.1. Motivación. Clases y Objetos

Criterios que se han ido incorporando a lo largo del tiempo en los estándares de la programación:

Programación estructurada (Structured programming) . Metodología de programación en la que la construcción de un algoritmo se basa en el uso de las estructuras de control vistas en el tema II, prohibiendo, entre otras cosas, el uso de estructuras de saltos arbitrarios del tipo `goto`.

Programación modular (Modular programming) : Cualquier metodología de programación que permita agrupar conjuntos de sentencias en *módulos* o *paquetes*.

Programación procedural (Procedural programming) . Metodología de programación modular en la que los módulos son las funciones.

Las funciones pueden usarse desde cualquier sitio tras su declaración (puede decirse que son **funciones globales (global functions)**) y se comunican con el resto del programa a través de sus parámetros y del resultado que devuelven (siempre que no se usen variables globales)

La base del diseño de una solución a un problema usando programación procedural consiste en analizar los procesos o tareas que ocurren en el problema e implementarlos usando funciones.

Nota. Lamentablemente, no hay un estándar en la nomenclatura usada. Muchos libros llaman programación modular a la programación con funciones. Nosotros usamos aquí el término modular en un sentido genérico. Otros libros llaman programación estructurada a lo que nosotros denominamos programación procedural.

Nota. No se usa el término **programación funcional (functional programming)** para referirse a la programación con funciones, ya que se acuñó para otro tipo de programación, a saber, un tipo de **programación declarativa (declarative programming)**

Programación orientada a objetos (Object oriented programming) (PDO). Metodología de programación modular en la que los módulos o paquetes se denominan **objetos (objects)** .

La base del diseño de una solución a un problema usando PDO consiste en analizar las entidades que intervienen en el problema e implementarlas usando objetos.

Un objeto aglutina en un único paquete datos y funciones. Las funciones incluidas en un objeto se denominan **métodos (methods)** . Los datos representan las características de una entidad y los métodos determinan su **comportamiento (behaviour)** .

Objeto: una ventana en Windows

- ▷ **Datos:** posición esquina superior izquierda, altura, anchura, está_minimizada
- ▷ **Métodos:** Minimiza(), Maximiza(), Agrandar(int tanto_por_ciento), etc.

Objeto: una cuenta bancaria

- ▷ **Datos:** identificador de la cuenta, saldo actual, descubierto que se permite
- ▷ **Métodos:** Ingresa(double cantidad), Retira(double cantidad), etc.

Objeto: un triángulo rectángulo

- ▷ **Datos:** los tres puntos que lo determinan A, B, C
- ▷ **Métodos:** ObtenerHipotenusa(), ObtenerSegmentoAB(), etc.

Objeto: una fracción

- ▷ **Datos:** Numerador y denominador
- ▷ **Métodos:** Simplifica(), Súmale(Fracción otra_fracción), etc.

Nota. Observad que un objeto es un dato **compuesto** e incluye otros datos y métodos.

Para construir un objeto, primero tenemos que definir su estructura. Esto se hace con el concepto de clase.

Una **clase (class)** es un **tipo de dato** definido por el programador. Se usa para representar una entidad.

Con la clase especificamos las características comunes y el comportamiento de una entidad. Es como un patrón o molde a partir del cual construimos los objetos.

Un **objeto (object)** es un **dato** cuyo tipo de dato es una clase. También se dirá que un objeto es la **instancia (instance)** de una clase.

```
class MiClase{
    ....
};
int main(){
    MiClase un_objeto_instancia_de_MiClase;
    MiClase otro_objeto_instancia_de_MiClase;
    .....
```

```
class CuentaBancaria{                // <- clase
    ....
};
int main(){
    CuentaBancaria una_cuenta;        // <- un objeto
    CuentaBancaria otra_cuenta;       // <- otro objeto
    .....
```

Los objetos **una_cuenta** y **otra_cuenta** existirán mientras esté ejecutándose **main**.

III.2.2. Encapsulación

- ▷ En Programación Procedural, la modularización se materializa al incluir en el mismo componente software (la función) un conjunto de instrucciones.
- ▷ En PDO, la modularización se materializa al incluir en el mismo componente software (la clase) los datos y los métodos (funciones que actúan sobre dichos datos). Este tipo de modularización se conoce como *encapsulación (encapsulation)*

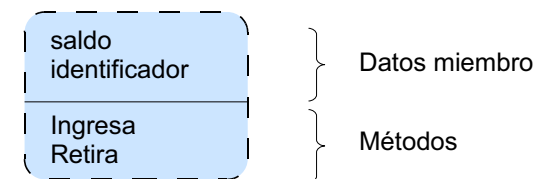
La encapsulación es un mecanismo de modularización.

Una clase se compone de:

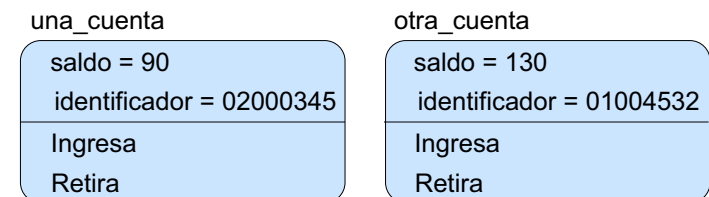
- ▷ *Datos miembro (data member)* :
Son las características que definen una entidad.
Todos los objetos que pertenecen a una misma clase tienen la misma estructura, pero cada objeto tiene un espacio en memoria distinto y por tanto unos valores propios para cada dato miembro. Diremos que el conjunto de valores específicos de los datos miembro en un momento determinado de un objeto conforman el *estado (state)* de dicho objeto.
- ▷ *Funciones miembro (member functions) o métodos (methods)* :
Son funciones definidas dentro de la clase.
Determinan el *comportamiento (behaviour)* de la entidad, es decir, el conjunto de operaciones que se pueden realizar sobre los objetos de la clase.
La definición de los métodos es la misma para todos los objetos.

```
class CuentaBancaria{           // <- clase
    ....
};
int main(){
    CuentaBancaria una_cuenta;   // <- un objeto
    CuentaBancaria otra_cuenta;  // <- otro objeto
    .....
```

Clase CuentaBancaria:



Objetos instancia de CuentaBancaria:



III.2.2.1. Datos miembro

Para declarar un dato miembro dentro de una clase, hay que especificar su **ámbito (scope)**, que podrá ser público o privado. Esto se indica con el **especificador de acceso (access specifier)** `public` o `private`.

Empezamos viendo el ámbito público.

Todas las declaraciones incluidas después de `public:` son públicas, es decir, accesibles desde fuera del objeto. Desde el `main` o desde otros objetos, accederemos a los datos públicos de los objetos a través del nombre del objeto, un punto y el nombre del dato.

```
class MiClase{
public:
    int dato;
};

int main(){
    MiClase un_objeto, otro_objeto;

    un_objeto.dato = 4;
    cout << un_objeto.dato;    // Imprime 4
    un_objeto.dato = 8;
    cout << un_objeto.dato;    // Imprime 8
    otro_objeto.dato = 7;
    cout << otro_objeto.dato;  // Imprime 7
```

Cada vez que modificamos un dato miembro, diremos que se ha modificado el estado del objeto.

Ejemplo. Cuenta bancaria.

Nota. Esta clase tiene problemas importantes de diseño que se irán arreglando a lo largo de este tema.

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

class CuentaBancaria{
public:
    double saldo;
    string identificador;
};

int main(){
    CuentaBancaria cuenta;
    string identificador_cuenta;
    double ingreso, retirada;

    cout << "\nIntroduce identificador a asignar a la cuenta:";
    cin >> identificador_cuenta;
    cuenta.identificador = identificador_cuenta;

    cout << "\nIntroduce cantidad inicial a ingresar: ";
    cin >> ingreso;

    cuenta.saldo = ingreso;

    cout << "\nIntroduce cantidad a ingresar: ";
    cin >> ingreso;

    cuenta.saldo = cuenta.saldo + ingreso;
```




```
cout << "\nIntroduce cantidad a retirar: ";
cin >> retirada;

cuenta.saldo = cuenta.saldo - retirada;

CuentaBancaria otra_cuenta; // Otro objeto de la clase CuentaBancaria

otra_cuenta.saldo = 30000; // <- No afecta al otro objeto cuenta

saldo = 300; // Error de compilación.
// saldo no es un dato definido en main.
```

Algunas aclaraciones:

▷ En vez de poner

```
cin >> identificador_cuenta;
cuenta.identificador = identificador_cuenta;
```

podríamos haber puesto directamente:

```
cin >> cuenta.identificador;
```

▷ Con las herramientas que conocemos, no puede leerse directamente un objeto por completo:

```
CuentaBancaria cuenta;

cin >> cuenta; // Error de compilación
```

Quando se crea un objeto, los datos miembro no tienen ningún valor asignado por defecto, a no ser que se inicialicen dentro de la clase. En este caso, al crear el objeto, se le asignarán automáticamente los valores especificados en la inicialización.

Desde C++11, la inicialización puede realizarse en la definición del dato miembro:

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

class CuentaBancaria{
public:
    double saldo = 0;
    string identificador = "";
};

int main(){
    CuentaBancaria cuenta; // "", 0

    cout << cuenta.saldo << "\n"; // 0
    cout << cuenta.identificador; // ""
    .....
```



348

Nota:

Realmente, C++ inicializa siempre cualquier `string` a `""` en el momento de su declaración, por lo que no es necesaria dicha inicialización.

III.2.2.2. Métodos

Por ahora, hemos conseguido crear varios objetos (cuentas bancarias) con sus propios datos miembros (saldo, identificador). Ahora vamos a ejecutar métodos sobre dichos objetos.

Definición

Los métodos determinan el comportamiento de los objetos de la clase. Son como funciones definidas dentro de la clase.

- ▷ El comportamiento de los métodos es el mismo para todos los objetos.

Al igual que ocurría con los datos miembro, podrán ser públicos o privados. Por ahora sólo consideramos métodos públicos.

Desde el `main` o desde otros objetos, accederemos a los métodos públicos de los objetos a través del nombre del objeto, un punto, el nombre del método y entre paréntesis los parámetros (en su caso).

```
class MiClase{
public:
    int dato;

    // dato = sqrt(5); Error de compilación. Las sentencias
    //                      deben estar dentro de los métodos

    void UnMetodo(){
        ....
    }
};

int main(){
    MiClase un_objeto;
```

```
un_objeto.dato = 4;
un_objeto.UnMetodo();
....
}
```

- ▷ No existe un consenso entre los distintos lenguajes de programación, a la hora de determinar el tipo de letra usado para las clases, objetos, métodos, etc. Nosotros seguiremos el siguiente:

- Tanto los identificadores de las clases como de los métodos empezarán con una letra mayúscula.

Si el nombre es compuesto usaremos la primera letra de la palabra en mayúscula: `CuentaBancaria`

- Los identificadores de los objetos, como cualquier otro dato, empezarán con minúscula.

Si el nombre es compuesto usaremos el símbolo de subrayado para separar los nombres: `mi_cuenta_bancaria`

Usaremos nombres para denotar las clases y verbos para los métodos.

- ▷ Los métodos pueden modificar el estado del objeto sobre el que actúan, es decir, pueden modificar los datos miembro. Acceden a ellos directamente.

Ejemplo. Añadimos métodos para ingresar y sacar dinero en la cuenta bancaria:

```
class CuentaBancaria{           // <- clase
public:
    double saldo = 0;
    string identificador; // C++ inicializa los string a ""

    void Ingresa(double cantidad){ // Dentro del método accedemos
        saldo = saldo + cantidad; // directamente al dato miembro
    }
    void Retira(double cantidad){
        saldo = saldo - cantidad;
    }
};

int main(){
    CuentaBancaria una_cuenta; // "", 0 un objeto
    CuentaBancaria otra_cuenta; // "", 0 otro objeto

    una_cuenta.identificador = "20310381450100006529"; // "2...9", 0
    una_cuenta.Ingresa(25); // "2...9", 25
    una_cuenta.Retira(10); // "2...9", 15
    .....
    otra_cuenta.identificador = "20310381450100007518"; // "2...8", 0
    otra_cuenta.Ingresa(45); // "2...8", 45
    otra_cuenta.Retira(15); // "2...8", 30
    .....
}
```



348

A destacar:

- ▷ Los métodos Ingresa y Retira acceden al dato miembro saldo por su nombre.

Los métodos acceden a los datos miembro directamente.

- ▷ Los métodos Ingresa y Retira modifican alguno de los datos miembro.

Los métodos pueden modificar el estado del objeto.

Ejercicio. Definamos la clase SegmentoDirigido

```
class SegmentoDirigido{
public:
    double x_1, y_1, x_2, y_2;
};

int main(){
    SegmentoDirigido un_segmento;

    cout << un_segmento.x_1; // Valor indeterminado

    un_segmento.x_1 = 3.4;
    un_segmento.y_1 = 5.6;
    un_segmento.x_2 = 4.5;
    un_segmento.y_2 = 2.3;

    cout << un_segmento.x_1; // 3.4
    cout << un_segmento.x_2; // 4.5
}
```



352

Si queremos, podemos inicializar los datos miembro en la definición de la clase:

```
class SegmentoDirigido{
public:
    double x_1 = 0.0,
           y_1 = 0.0,
           x_2 = 0.0,
           y_2 = 0.0;
};

int main(){
    SegmentoDirigido un_segmento;

    cout << un_segmento.x_1; // 0.0
}
```



352

Ejercicio. Definir sendos métodos TrasladaHorizontal y TrasladaVertical para trasladar un segmento un número de unidades.

```
class SegmentoDirigido{
public:
    double x_1 = 0.0,
           y_1 = 0.0,
           x_2 = 0.0,
           y_2 = 0.0;

    void TrasladaHorizontal(double unidades){
        x_1 = x_1 + unidades;
        x_2 = x_2 + unidades;
    }

    void TrasladaVertical(double unidades){
        y_1 = y_1 + unidades;
        y_2 = y_2 + unidades;
    }
};

int main(){
    SegmentoDirigido un_segmento;

    un_segmento.x_1 = 3.4;
    un_segmento.y_1 = 5.6;
    un_segmento.x_2 = 4.5;
    un_segmento.y_2 = 2.3;

    un_segmento.TrasladaHorizontal(10);

    cout << un_segmento.x_1; // 13.4
    cout << un_segmento.x_2; // 14.5
}
```



352

Ejercicio. Calcular la longitud de un segmento dirigido.

```
class SegmentoDirigido{
public:
    double x_1 = 0.0,
           y_1 = 0.0,
           x_2 = 0.0,
           y_2 = 0.0;
    double Longitud(){
        double resta_abscisas = x_2 - x_1;
        double resta_ordenadas = y_2 - y_1;

        return sqrt(resta_abscisas * resta_abscisas +
                    resta_ordenadas * resta_ordenadas);
    }
    void TrasladaHorizontal(double unidades){
        x_1 = x_1 + unidades;
        x_2 = x_2 + unidades;
    }
    void TrasladaVertical(double unidades){
        y_1 = y_1 + unidades;
        y_2 = y_2 + unidades;
    }
};

int main(){
    SegmentoDirigido un_segmento;

    un_segmento.x_1 = 3.4;
    un_segmento.y_1 = 5.6;
    un_segmento.x_2 = 4.5;
    un_segmento.y_2 = 2.3;

    cout << "\nLongitud del segmento = " << un_segmento.Longitud();
}
```



Para representar las clases se utiliza una notación gráfica con una caja que contiene el nombre de la clase. A continuación se incluye un bloque con los datos miembro y por último un tercer bloque con los métodos. Los métodos públicos se notarán con un símbolo + (posteriormente se verán los privados)

CuentaBancaria	
+ double	saldo
+ string	identificador
+ void	Ingresa(double cantidad)
+ void	Retira(double cantidad)



348

SegmentoDirigido	
+ double	x_1
+ double	y_1
+ double	x_2
+ double	y_2
+ double	Longitud()
+ void	TrasladaHorizontal(double unidades)
+ void	TrasladaVertical(double unidades)



352

Controlando el acceso a los datos miembro

Ya vimos que si al llamar a una función siempre debemos realizar una serie de instrucciones, éstas deberían ir dentro de la función (página 309) Con los métodos pasa lo mismo.

Ejemplo. En el ejemplo de la cuenta bancaria, ¿cómo implementamos una restricción real como que, por ejemplo, los ingresos sólo puedan ser positivos y las retiradas de fondos inferiores al saldo? ¿Lo comprobamos en el main antes de llamar a los métodos?

```
class CuentaBancaria{
public:
    double saldo          = 0;
    string identificador; // C++ inicializa los string a ""

    void Ingresa(double cantidad){
        saldo = saldo + cantidad;
    }
    void Retira(double cantidad){
        saldo = saldo - cantidad;
    }
};

int main(){
    CuentaBancaria cuenta;
    double ingreso, retirada;

    cuenta.identificador = "20310381450100006529";    // "2...9", 0

    cin >> ingreso;

    if (ingreso > 0)
        cuenta.Ingresa(ingreso);
```



348

```
cin >> retirada;

if (retirada > 0 && retirada <= cuenta.saldo)
    cuenta.Retira(retirada);
```



Cada vez que realicemos un ingreso o retirada de fondos, habría que realizar las anteriores comprobaciones, por lo que es propenso a errores.

Solución: Lo programamos dentro del método correspondiente. Siempre que se ejecute el método se realizará la comprobación automáticamente.

```
class CuentaBancaria{  
public:  
    double saldo          = 0;  
    string identificador; // C++ inicializa los string a ""  
  
    void Ingresa(double cantidad){ 😊  
        if (cantidad > 0)  
            saldo = saldo + cantidad;  
    }  
  
    void Retira(double cantidad){ 😊  
        if (cantidad > 0 && cantidad <= saldo)  
            saldo = saldo - cantidad;  
    }  
};  
  
int main(){  
    CuentaBancaria cuenta; // "", 0  
  
    cuenta.identificador = "20310381450100006529"; // "2...9", 0  
    cuenta.Ingresa(-3000); // "2...9", 0  
    cuenta.Ingresa(25);    // "2...9", 25  
    cuenta.Retira(-10);    // "2...9", 25  
    cuenta.Retira(10);     // "2...9", 15  
    cuenta.Retira(100);    // "2...9", 15  
}
```



348

Los métodos permiten establecer la política de acceso a los datos miembro, es decir, determinar cuáles son las operaciones permitidas con ellos.

Llamadas entre métodos dentro del propio objeto

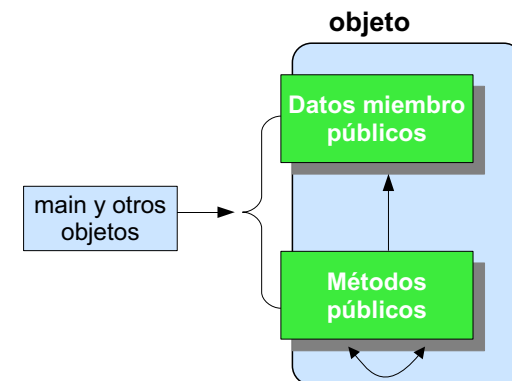
Ya sabemos cómo ejecutar los métodos sobre un objeto:

```
objeto.UnMetodo(...);
```

¿Pero pueden llamarse dentro del objeto unos métodos a otros? Si.

Todos los métodos de un objeto pueden llamar a los métodos del mismo objeto. La llamada se especifica indicando el nombre del método. No hay que anteponer el nombre de ningún objeto ya que en la definición de la clase no tenemos ningún objeto destacado.

```
class MiClase{  
public:  
    void UnMetodo(parámetros formales){  
    }  
    void OtroMetodo(...){  
        UnMetodo(parámetros actuales);  
    }  
};
```



Ejemplo. Implementar el método `Traslada` para que traslade el segmento tanto en horizontal como en vertical.

```
class SegmentoDirigido{
public:
    double x_1 = 0.0,
           y_1 = 0.0,
           x_2 = 0.0,
           y_2 = 0.0;

    void TrasladaHorizontal(double unidades){
        x_1 = x_1 + unidades;
        x_2 = x_2 + unidades;
    }

    void TrasladaVertical(double unidades){
        y_1 = y_1 + unidades;
        y_2 = y_2 + unidades;
    }

    // Traslada en Horizontal y en Vertical

    void TrasladaRepitiendoCodigo (double und_horiz, double und_vert){
        x_1 = x_1 + und_horiz;
        x_2 = x_2 + und_horiz;
        y_1 = y_1 + und_vert;
        y_2 = y_2 + und_vert;
    }

    void Traslada(double und_horiz, double und_vert){
        TrasladaHorizontal(und_horiz);
        TrasladaVertical(und_vert);
    }
    .....
};
```



```
int main(){
    SegmentoDirigido un_segmento;

    un_segmento.x_1 = 3.4;
    un_segmento.y_1 = 5.6;
    un_segmento.x_2 = 4.5;
    un_segmento.y_2 = 2.3;

    cout << "Segmento Dirigido.\n\n";
    cout << "Antes de la traslación:\n";
    cout << un_segmento.x_1 << " , " << un_segmento.y_1;    // 3.4 , 5.6
    cout << "\n";
    cout << un_segmento.x_2 << " , " << un_segmento.y_2;    // 4.5 , 2.3



    un_segmento.Traslada(5.0, 10.0);

    cout << "\n\nDespués de la traslación:\n";
    cout << un_segmento.x_1 << " , " << un_segmento.y_1;    // 8.4 , 15.6
    cout << "\n";
    cout << un_segmento.x_2 << " , " << un_segmento.y_2;    // 9.5 , 12.3
}
```

http://decsai.ugr.es/~carlos/FP/III_segmento_dirigido_todo_public.cpp

Dentro de la clase, los métodos pueden llamarse unos a otros. Esto nos permite cumplir el principio de una única vez.

Ejercicio. Aplicad un interés porcentual al saldo de la cuenta bancaria.

```
class CuentaBancaria{  
public:  
    double saldo          = 0;  
    string identificador; // C++ inicializa los string a ""  
  
    void AplicaInteresPorcentualRepitiendoCodigo(int tanto_porcento){  
        double cantidad;  
        cantidad = saldo * tanto_porcento / 100.0;  
  
        if (cantidad > 0)   
            saldo = saldo + cantidad;  
    }  
  
    void AplicaInteresPorcentual (int tanto_porcento){   
        Ingresar (saldo * tanto_porcento / 100.0);  
    }  
  
    void Ingresar(double cantidad){  
        if (cantidad > 0)  
            saldo = saldo + cantidad;  
    }  
  
    void Retira(double cantidad){  
        if (cantidad > 0 && cantidad <= saldo)  
            saldo = saldo - cantidad;  
    }  
};  
  
int main(){  
    CuentaBancaria cuenta; // "", 0  
  
    cuenta.identificador = "20310381450100006529"; // "2...9", 0  
    una_cuenta.Ingresar(25); // "2...9", 25  
    una_cuenta.AplicaInteresPorcentual(3); // "2...9", 25.75  
}
```



http://decsai.ugr.es/~carlos/FP/III_cuenta_bancaria_todo_public.cpp



III.2.3. Ocultación de información**III.2.3.1. Ámbito público y privado**

- ▷ En Programación Procedural, la ocultación de información se consigue con el ámbito local a la función (datos locales y parámetros formales).
- ▷ En PDO, la ocultación de información se consigue con el ámbito local a los métodos (datos locales y parámetros formales) y además con el ámbito `private` en las clases, tanto en los datos miembro como en los métodos.

Recuperemos el ejemplo de la cuenta bancaria. Tenemos dos formas de ingresar 25 euros:

```
cuenta.Ingresar(25);  
cuenta.saldo = cuenta.saldo + 25;
```

¿Y si hubiésemos puesto -3000?

```
int main()  
{  
    CuentaBancaria cuenta; // "", 0  
  
    cuenta.identificador = "20310381450100006529"; // "2...9", 0  
  
    cuenta.Ingresar(-3000); // "2...9", 0   
  
    cuenta.saldo = -3000; // "2...9", -3000 
```

Para poder controlar las operaciones que están permitidas sobre el saldo, debemos usar el método `Ingresar`. ¿Cómo imponemos que siempre sea así y que no podamos acceder directamente a él con `cuenta.saldo`? Haciendo que `saldo` sólo sea accesible desde dentro de la clase, es decir, que tenga ámbito privado.

Al declarar los miembros de una clase, se debe indicar su ámbito es decir, desde dónde se van a poder utilizar:

▷ **Ámbito público (Public scope)** .

- Se indica con el especificador de ámbito `public`
- Los miembros públicos son visibles dentro y fuera del objeto.

▷ **Ámbito privado (Private scope)** .

- Se indica con el especificador de ámbito `private` (éste es el ámbito por defecto si no se pone nada)

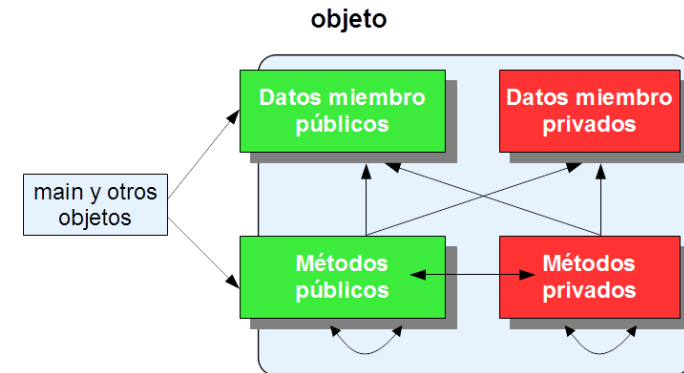
```
class Clase{
private:
    int dato_privado;
    void MetodoPrivado(){
        .....
    }
public:
    int dato_publico;
    void MetodoPublico(){
        .....
    }
};
```

- Los miembros privados sólo se pueden usar desde dentro del objeto. No son accesibles desde fuera.

```
int main(){
    Clase objeto;

    objeto.dato_publico = 4;
    objeto.dato_privado = 4;    // Error compilación
    objeto.MetodoPrivado();    // Error compilación
```

- ▷ Los métodos (públicos o privados) del objeto acceden a los métodos y datos miembro (públicos o privados) por su nombre.



En la representación gráfica de las clases, los datos miembro privados se notan con signo `-`. La gráfica de la izquierda sería una representación con el estándar **UML** (Unified Modeling Language) *original*. Nosotros usaremos una adaptación como aparece a la derecha, semejante a la sintaxis de C++.

Clase
- dato_privado: int + dato_publico: double
- MetodoPrivado(param: double): int + MetodoPublico(param: int): void

UML puro



Clase
- int dato_privado + double dato_publico
- int MetodoPrivado(double param) + void MetodoPublico(int param)

UML adaptado

III.2.3.2. Datos miembro privados

Supongamos que cambiamos el ámbito público de los datos miembro de la clase Cuenta_Bancaria a privado.

```
class CuentaBancaria{
private:
    double saldo          = 0;
    string identificador; // C++ inicializa los string a ""
public:
    void Ingresa(double cantidad){
        if (cantidad > 0)
            saldo = saldo + cantidad;
    }
    void Retira(double cantidad){
        if (cantidad > 0 && cantidad <= saldo)
            saldo = saldo - cantidad;
    }
    void AplicaInteresPorcentual(int tanto_porcento){
        Ingresa (saldo * tanto_porcento / 100.0);
    }
};

int main(){
    CuentaBancaria cuenta;
```



Las siguientes sentencias provocan un error de compilación ya que los datos miembro son ahora privados:

```
cuenta.identificador = "20310381450100006529";
cout << cuenta.saldo;
```

¿Cómo le asignamos entonces un valor?

Al trabajar con datos miembro privados debemos añadirle a la clase:

- ▷ Métodos para modificar los datos miembro (aquellos que se deseen modificar desde fuera)
- ▷ Métodos para obtener el valor actual de los datos miembro (aquellos a los que se desee acceder desde fuera)

```
class CuentaBancaria{
private:
    double saldo          = 0.0;
    string identificador; // C++ inicializa los string a ""
public:
    void SetIdentificador(string identificador_cuenta){
        if (identificador_cuenta.size() == 20 &&
            identificador_cuenta.substr(0,4) == "2031")

            identificador = identificador_cuenta;
    }
    string Identificador(){
        return identificador;
    }
    void SetSaldo(double cantidad){
        if (cantidad > 0)
            saldo = cantidad;
    }
    double Saldo(){
        return saldo;
    }
    void Ingresa(double cantidad){
        if (cantidad > 0)
            saldo = saldo + cantidad;
    }
    void Retira(double cantidad){
```



```

        if (cantidad > 0 && cantidad <= saldo)
            saldo = saldo - cantidad;
    }
    void AplicaInteresPorcentual(int tanto_porcento){
        Ingresa (saldo * tanto_porcento / 100.0);
    }
};
int main(){
    CuentaBancaria cuenta;    // "", 0

    cuenta.SetIdentificador("inválido");           // "", 0
    cuenta.SetIdentificador("20310381450100006529"); // "2...9", 0
    cuenta.Ingresa(25);                             // "2...9", 25
    cuenta.SetSaldo(-300);                           // "2...9", 25

    cout << cuenta.Saldo();           // <- Paréntesis, pues es un método
}

```

Parece razonable no incluir un método `SetSaldo` pues los cambios en el saldo deberían hacerse siempre con `Ingresa` y `Retira`. Para ello, basta eliminar `SetSaldo`.

http://decsai.ugr.es/~carlos/FP/III_cuenta_bancaria_datos_miembro_private.cpp

De nuevo vemos que los métodos permiten establecer la política de acceso a los datos miembro. La clase nos quedaría así:

CuentaBancaria	
- double	saldo
- string	identificador
+ void	SetIdentificador(string identificador_cuenta)
+ string	Identificador()
+ double	Saldo()
+ void	Ingresa(double cantidad)
+ void	Retira(double cantidad)
+ void	AplicaInteresPorcentual(int tanto_por_ciento)



Salvo casos muy justificados, no definiremos datos miembro públicos. Siempre serán privados.

El acceso a ellos desde fuera de la clase se hará a través de métodos públicos de la clase.



Interfaz (Interface) de una clase: Es el conjunto de datos y métodos públicos de dicha clase. Como usualmente los datos son privados, el término interfaz suele referirse al conjunto de métodos públicos.

Ejemplo. Definid la clase `Fecha` que representa un día, mes y año. Decidimos definir un único método para cambiar los tres valores a la misma vez, en vez de tres métodos independientes.

Fecha	
- int	dia
- int	mes
- int	anio
+ void	SetDiaMesAnio (int el_dia, int el_mes, int el_anio)
+ int	GetDia()
+ int	GetMes()
+ int	GetAnio()
+ string	ToString()

```
class Fecha {
private:
    int dia = 0,
        mes = 0,
        anio = 0;
public:
    void SetDiaMesAnio(int el_dia, int el_mes, int el_anio){
        if (el_anio >= 1850 && el_anio <= 3000 &&
            1 <= el_dia && el_dia <= 31 &&
            1 <= el_mes && el_mes <= 12){

            dia = el_dia;
            mes = el_mes;
            anio = el_anio;
        }
    }
    int GetDia(){
        return dia;
    }
    int GetMes(){
        return mes;
    }
}
```

```
    }
    int GetAnio(){
        return anio;
    }
    string ToString(){
        return to_string(dia) + "/" + to_string(mes)
            + "/" + to_string(anio);
    }
};

int main(){
    Fecha nacimiento_JC;           // 0, 0, 0
    Fecha otra_fecha;              // 0, 0, 0

    nacimiento_JC.SetDiaMesAnio(27, 2, 1967); // 27, 2, 1967
    nacimiento_JC.SetDiaMesAnio(-8, 2, 1967); // 27, 2, 1967

    cout << nacimiento_JC.ToString();

    otra_fecha.SetDiaMesAnio(8, -2, 1967);     // 0, 0, 0
}
```

http://decsai.ugr.es/~carlos/FP/III_fecha.cpp

La comprobación sobre el día, mes y año debe mejorarse ya que no todos los meses traen 31 días. Lo haremos posteriormente.

Ejercicio. Cambiemos el ámbito (de public a private) de los datos miembro de la clase `SegmentoDirigido`. Obligamos cambiar los 4 datos simultáneamente.

SegmentoDirigido	
- double	x_1
- double	y_1
- double	x_2
- double	y_2
+ void	SetCoordenadas (double origen_abscisa, double origen_ordenada, double final_abscisa, double final_ordenada)
+ double	OrigenAbscisa()
+ double	OrigenOrdenada()
+ double	FinalAbscisa()
+ double	FinalOrdenada()
+ double	Longitud()
+ void	TrasladaHorizontal(double unidades)
+ void	TrasladaVertical(double unidades)
+ void	Traslada(double en_horizontal, double en_vertical)

```
class SegmentoDirigido{
private:
    double x_1 = 0.0,
           y_1 = 0.0,
           x_2 = 0.0,
           y_2 = 0.0;
public:
    void SetCoordenadas(double origen_abscisa,
                        double origen_ordenada,
                        double final_abscisa,
                        double final_ordenada){
        if (! (origen_abscisa == final_abscisa &&
                origen_ordenada == final_ordenada)){
            x_1 = origen_abscisa;
            y_1 = origen_ordenada;
            x_2 = final_abscisa;
            y_2 = final_ordenada;
        }
    }
    double OrigenAbscisa(){
        return x_1;
    }
    double OrigenOrdenada(){
        return y_1;
    }
    double FinalAbscisa(){
        return x_2;
    }
    double FinalOrdenada(){
        return y_2;
    }
    // Los métodos Longitud, Traslada, TrasladaHorizontal
    // y TrasladaVertical no varían
}
```

```
};

int main(){
    SegmentoDirigido un_segmento;    // 0.0, 0.0, 0.0, 0.0

    un_segmento.SetCoordenadas(3.4, 5.6, 4.5, 2.3);

    cout << "Segmento Dirigido.\n\n";
    cout << "Antes de la traslación:\n";
    cout << un_segmento.OriginAbscisa() << " , "
        << un_segmento.OriginOrdenada();           // 3.4 , 5.6
    cout << "\n";
    cout << un_segmento.FinalAbscisa() << " , "
        << un_segmento.FinalOrdenada();             // 4.5 , 2.3

    un_segmento.Traslada(5.0, 10.0);

    cout << "\n\nDespués de la traslación:\n";
    cout << un_segmento.OriginAbscisa() << " , "
        << un_segmento.OriginOrdenada();           // 8.4 , 15.6
    cout << "\n";
    cout << un_segmento.FinalAbscisa() << " , "
        << un_segmento.FinalOrdenada();             // 9.5 , 12.3

    SegmentoDirigido otro_segmento;                // 0.0, 0.0, 0.0, 0.0
    otro_segmento.SetCoordenadas(3.4, 5.6, 3.4, 5.6); // 0.0, 0.0, 0.0, 0.0
    .....
```

Ejercicio. Representemos una circunferencia.

Circunferencia	
- double	centro_x
- double	centro_y
- double	radio
+ void	SetCentro(double abscisa, double ordenada)
+ void	SetRadio(double el_radio)
+ double	GetAbscisaCentro()
+ double	GetOrdenadaCentro()
+ double	GetRadio()
+ double	Longitud()
+ double	Area()
+ void	Traslada(double en_horizontal, double en_vertical)

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

```
class Circunferencia{
private:
    double centro_x = 0.0;
    double centro_y = 0.0;
    double radio    = 0.0;
public:
    void SetCentro(double abscisa, double ordenada){
        centro_x = abscisa;
        centro_y = ordenada;
    }
    void SetRadio(double el_radio){
        radio = el_radio;
    }
    double GetAbscisaCentro(){
        return centro_x;
    }
    double GetOrdenadaCentro(){
        return centro_y;
    }
}
```

```
double GetRadio(){
    return radio;
}
double Longitud(){
    return 2*PI*radio;
}
double Area(){
    return PI*radio*radio;
}
void Traslada(double en_horizontal, double en_vertical){
    centro_x = centro_x + en_horizontal;
    centro_y = centro_y + en_vertical;
}
};

int main(){
    Circunferencia mi_aro;
    double longitud_mi_aro;

    mi_aro.SetCentro(4.5, 6.7);
    mi_aro.SetRadio(2.1);
    longitud_mi_aro = mi_aro.Longitud();    // 13.19468

    mi_aro.SetRadio(8.3);                  // Cambiamos el radio
    longitud_mi_aro = mi_aro.Longitud();    // 52,15044

    mi_aro.Traslada(10.1, 15.2);           // Traslación
    longitud_mi_aro = mi_aro.Longitud();    // 52,15044
```

http://decsai.ugr.es/~carlos/FP/III_circunferencia.cpp

III.2.3.3. Métodos privados

Si tenemos que realizar un mismo conjunto de operaciones en varios sitios de la clase, usaremos un método para así no repetir código. Si no queremos que se pueda usar desde fuera de la clase, lo declaramos `private`.

Ejemplo. Sobre la clase `CuentaBancaria`, supongamos que no permitimos saldos superiores a 10000 euros. Para ello, definimos el método privado `EsCorrecto` que realiza las comprobaciones pertinentes. Llamamos a este método desde `Ingresar` y `Retira`.

```
class CuentaBancaria{
private:
    double saldo          = 0.0;
    string identificador;  // C++ inicializa los string a ""

    bool EsCorrectoSaldo(double saldo_propuesto){
        return saldo_propuesto >= 0 && saldo_propuesto <= 10000;
    }
public:
    void SetIdentificador(string identificador_cuenta){
        identificador = identificador_cuenta;
    }
    string Identificador(){
        return identificador;
    }
    double Saldo(){
        return saldo;
    }
    void Ingresar(double cantidad){
        double saldo_resultante;

        if (cantidad > 0){
```



```

        saldo_resultante = saldo + cantidad;

        if (EsCorrectoSaldo (saldo_resultante))
            saldo = saldo_resultante;
    }
}

void Retira(double cantidad){
    double saldo_resultante;

    if (cantidad > 0){
        saldo_resultante = saldo - cantidad;

        if (EsCorrectoSaldo (saldo_resultante))
            saldo = saldo_resultante;
    }
}

void AplicaInteresPorcentual(int tanto_porcentaje){
    Ingresa (saldo * tanto_porcentaje / 100.0);
}

};

int main(){
    CuentaBancaria cuenta;    // saldo = 0

    cuenta.Ingresa(50000);    // saldo = 0
    cuenta.Ingresa(8000);    // saldo = 8000

    bool es_correcto;
    es_correcto = cuenta.EsCorrectoSaldo(50000); // Error de compilación
                                                // Método private

```

Podemos comprobar que se repite el código siguiente:

```

if (EsCorrectoSaldo (saldo_resultante))
    saldo = saldo_resultante;

```

Si se desea, puede definirse el siguiente método privado:

```

void SetSaldo (double saldo_propuesto){
    if (EsCorrectoSaldo (saldo_propuesto))
        saldo = saldo_propuesto;
}

```

De forma que el método Ingresa, por ejemplo, quedaría así:

```

void Ingresa(double cantidad){
    double saldo_resultante;

    if (cantidad > 0){
        saldo_resultante = saldo + cantidad;
        SetSaldo(saldo_resultante);
    }
}

```

Observad que en el caso de que el saldo propuesto sea incorrecto, se deja el valor antiguo. Este criterio es correcto, aunque existen otras alternativas, tal y como se verá posteriormente.

Recordemos que SetSaldo no queríamos que fuese public ya que sólo permitíamos ingresos y retiradas de fondos y no asignaciones directas.

En cuanto al identificador, introducimos también un método privado que compruebe que es correcto, y lo llamamos desde SetIdentificador (éste sí es public)

La clase nos quedaría así:

CuentaBancaria	
- double	saldo
- double	identificador
- bool	EsCorrectoSaldo(double saldo_propuesto)
- bool	EsCorrectoIdentificador(string identificador_propuesto)
- void	SetSaldo(double saldo_propuesto)
+ void	SetIdentificador(string identificador_cuenta)
+ string	Identificador()
+ double	Saldo()
+ void	Ingresa(double cantidad)
+ void	Retira(double cantidad)
+ void	AplicaInteresPorcentual(int tanto_porcentaje)

http://decsai.ugr.es/~carlos/FP/III_cuenta_bancaria_metodos_privados.cpp

Los métodos privados se usan para realizar tareas propias de la clase que no queremos que se puedan invocar desde fuera de ésta.

Ejercicio. Reescribid el ejemplo de la Fecha de la página 349 para que la comprobación de que la fecha es correcta se haga en un método privado.

Fecha	
- int	dia
- int	mes
- int	anio
+ void	EsFechaCorrecta(int el_dia, int el_mes, int el_anio)
+ void	SetDiaMesAnio(int el_dia, int el_mes, int el_anio)
+ int	GetDia()
+ int	GetMes()
+ int	GetAnio()
+ string	ToString()

```
class Fecha {
private:
    int dia = 0,
        mes = 0,
        anio = 0;

    bool EsFechaCorrecta(int el_dia, int el_mes, int el_anio){
        return el_anio >= 1850 && el_anio <= 3000 &&
            1 <= el_dia && el_dia <= 31 &&
            1 <= el_mes && el_mes <= 12;
    }

public:
    void SetDiaMesAnio(int el_dia, int el_mes, int el_anio){
        if (EsFechaCorrecta(el_dia, el_mes, el_anio)){
            dia = el_dia;
            mes = el_mes;
            anio = el_anio;
        }
    }
    .....
};
```

Ejercicio. Reescribid el ejemplo del segmento de la página 351 para que la comprobación de que las coordenadas son correctas se hagan en un método privado.

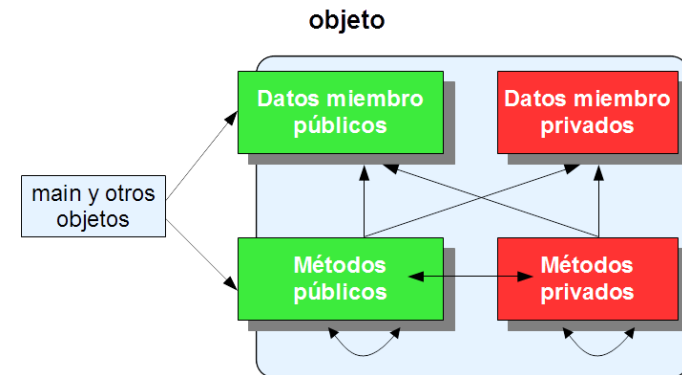
```
class SegmentoDirigido{
private:
    double x_1 = 0.0,
           y_1 = 0.0,
           x_2 = 0.0,
           y_2 = 0.0;

    bool SonCorrectas(double abs_1, ord_1, abs_2, ord_2){
        return !(abs_1 == abs_2 && ord_1 == ord_2);
    }

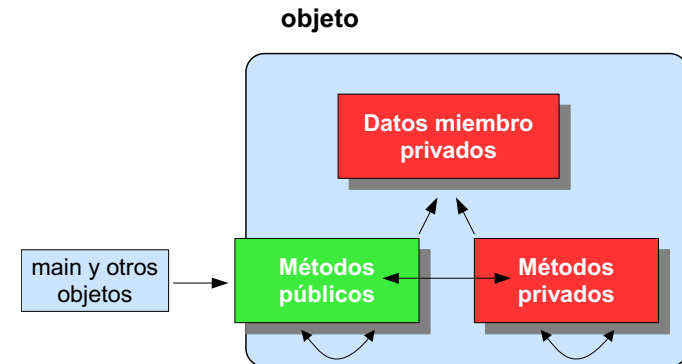
public:
    void SetCoordenadas(double origen_abscisa,
                       double origen_ordenada,
                       double final_abscisa,
                       double final_ordenada){
        if (SonCorrectas(origen_abscisa, origen_ordenada,
                        final_abscisa, final_ordenada)){
            x_1 = origen_abscisa;
            y_1 = origen_ordenada;
            x_2 = final_abscisa;
            y_2 = final_ordenada;
        }
    }
    .....
};
```

http://decsai.ugr.es/~carlos/FP/III_segmento_dirigido_metodos_privados.cpp

En resumen. Lo que C++ permite:



Lo que nosotros haremos:



III.2.4. Estado inválido de un objeto

¿Qué ocurre si una variable no tiene asignado un valor? Cualquier operación que realicemos con ella devolverá un valor indeterminado.

```
int main(){
    int euros_enteros;    // euros_enteros = ?
    cout << euros_enteros; // Imprime un valor indeterminado
```

Lo mismo ocurre con un objeto que contenga datos miembro sin un valor determinado.

```
class Clase{
private:
    int dato;
public:
    void SetDato(int valor){
        dato = valor;
    }
    int GetDato(){
        return dato;
    }
};

int main(){
    Clase objeto;           // objeto.dato = ?
    cout << objeto.GetDato(); // Imprime un valor indeterminado
```

Diremos que un objeto se encuentra en un momento dado en un **estado inválido** si algunos de sus datos miembros **esenciales** no tienen un valor correcto.

¿Qué entendemos por correcto? Depende de cada clase. Desde luego, si es un valor indeterminado (no asignado previamente), será incorrecto.

Ejemplo. Definid la clase Dinero.

Dinero	
- int	euros_enteros
- int	centimos_enteros
+ void	SetEurosDecimal(double euros)
+ void	SetEuros_y_Centimos(int numero_euros, int numero_centimos)
+ int	EurosEnteros()
+ int	CentimosEnteros()

```
class Dinero{
private:
    int euros_enteros;
    int centimos_enteros;
public:
    int EurosEnteros(){
        return euros_enteros;
    }
    int CentimosEnteros(){
        return centimos_enteros;
    }
    // Evitar en la medida de lo posible la asignación a través
    // de SetEurosDecimal ya que la representación con double
    // no garantiza ser exacta.
    // Sólo se consideran los dos primeros decimales

    void SetEurosDecimal(double euros){
        int euros_enteros_propuestos, centimos_enteros_propuestos;

        euros_enteros_propuestos = euros; // casting a int
        centimos_enteros_propuestos = 100 * (euros - euros_enteros);

        SetEuros_y_Centimos(euros_enteros_propuestos,
                             centimos_enteros_propuestos);
    }
}
```

```
void SetEuros_y_Centimos(int numero_euros, int numero_centimos){
    if (numero_euros > 0 && numero_centimos > 0){
        centimos_enteros = numero_centimos % 100;
        euros_enteros    = numero_euros + numero_centimos / 100;
    }
}
};
int main(){
    Dinero dinero; // ? euros, ? céntimos. Estado inválido

    dinero.SetEuros_y_Centimos(200, 150); // 200, 150
                                           // Estado válido
}
```

Para que el objeto `Dinero` esté en un estado válido en el mismo momento de la declaración, podemos usar la inicialización de los datos miembro dentro de la clase. En este ejemplo podemos admitir que un objeto dinero con 0 euros y 0 céntimos está en un estado válido, es decir, es una cantidad de dinero real:

```
class Dinero{
private:
    int euros_enteros = 0;
    int centimos      = 0;
public:
    .....
};
int main(){
    Dinero dinero; // 0 euros, 0 céntimos. Estado válido
}
```

http://decsai.ugr.es/~carlos/FP/III_dinero.cpp

Ejemplo. Definid la clase `Persona`. Representaremos su nombre, altura y edad.

Persona	
- string	nombre
- int	edad
- int	altura
- bool	EsCorrectoNombre(string un_nombre)
- bool	EsCorrectaEdad(string una_edad)
- bool	EsCorrectaAltura(string una_altura)
+ void	SetNombre(string nombre_persona)
+ void	SetEdad(int edad_persona)
+ void	SetAltura(int altura_persona)
+ string	Nombre()
+ int	Edad()
+ int	Altura()

```
class Persona{
private:
    string nombre;
    int edad;
    int altura;

    bool EsCorrectoNombre(string un_nombre){
        return un_nombre.size() < 100;
    }
    bool EsCorrectaEdad(int una_edad){
        return una_edad > 0 && una_edad < 120;
    }
    bool EsCorrectaAltura(int una_altura){
        return una_altura > 50 && una_altura < 250;
    }
public:
    string Nombre(){
        return nombre;
    }
}
```

```
int Edad(){
    return edad;
}
int Altura(){
    return altura;
}
void SetNombre(string nombre_persona){
    if (EsCorrectoNombre(nombre_persona))
        nombre = nombre_persona;
}
void SetEdad(int edad_persona){
    if (EsCorrectaEdad(edad_persona))
        edad = edad_persona;
}
void SetAltura(int altura_persona){
    if (EsCorrectaAltura(altura_persona))
        altura = altura_persona;
}
};

int main(){
    Persona una_persona; // nombre = ?, edad = ?, altura = ?
                          // Estado inválido

    una_persona.SetNombre("Juan");
    una_persona.SetEdad(27);
    una_persona.SetAltura(180);

    // una_persona -> "Juan", 27, 180. Estado válido
```

Podríamos inicializar los datos miembro dentro de la clase, pero tendríamos que elegir valores arbitrarios, por lo que, igualmente, también son considerados incorrectos.

```
class Persona{
private:
    string nombre = "Pedro"; // No tiene sentido un valor concreto
    int edad = 25; // No tiene sentido un valor concreto
    int altura = 150; // No tiene sentido un valor concreto
    .....
};

int main(){
    Persona una_persona; // Pedro, 25, 150 son valores arbitrarios.
                          // una_persona está en un estado inválido
```

Como mal menor, podemos inicializar los datos miembro a un dato concreto que indique que todavía no tienen asignado un valor correcto (en caso de que sea posible seleccionar tal dato)

```
class Persona{
private:
    string nombre; // C++ inicializa los string a ""
    int edad = 0;
    int altura = 0;
public:
    .....
};

int main(){
    Persona una_persona; // "", 0, 0. Estado inválido

    cout << una_persona.Edad(); // Imprime 0
```

El que un objeto esté en un estado válido en un momento determinado, depende de los valores que tenga en ese momento sus datos miembros y de la semántica de éstos.

En el caso de que tenga sentido, fomentaremos la inicialización de los datos miembros dentro de la clase.

¿Cómo garantizamos que un objeto esté siempre en un estado válido en el mismo momento que se crea? Lo haremos usando los constructores.

III.2.5. Constructores

III.2.5.1. Definición de constructores

Un **constructor** (*constructor*) es como un método sin tipo (`void`) de la clase en el que se incluyen todas las acciones que queramos realizar en el momento de construir un objeto.

- ▷ El constructor se define dentro de la clase, en la sección `public`.
- ▷ Debe llamarse obligatoriamente, igual que la clase. No se pone `void`, sino únicamente el nombre de la clase.
- ▷ Cuando se cree un objeto, el compilador ejecutará las instrucciones especificadas en el constructor.
- ▷ Se pueden incluir parámetros, en cuyo caso, habrá que incluir los correspondientes parámetros actuales en el momento de la definición del objeto. En caso contrario, se produce un error de compilación.

Ejemplo. Sobre la clase `Persona`, añadimos un constructor para establecer los valores de los datos miembro, en el mismo momento de la creación del objeto.

Persona	
- string	nombre
- int	edad
- int	altura
- bool	EsCorrectoNombre(string un_nombre)
- bool	EsCorrectaEdad(string una_edad)
- bool	EsCorrectaAltura(string una_altura)
+	Persona (string el_nombre, int la_edad, int la_altura)
+	void SetNombre(string nombre_persona)
+	void SetEdad(int edad_persona)
+	void SetAltura(int altura_persona)
+	string Nombre()
+	int Edad()
+	int Altura()

```
class Persona{
private:
    string nombre;    // C++ inicializa los string a ""
    int edad         = 0;
    int altura        = 0;

    bool EsCorrectoNombre(string un_nombre){
        return un_nombre.size() < 100;
    }
    bool EsCorrectaEdad(int una_edad){
        return una_edad > 0 && una_edad < 120;
    }
    bool EsCorrectaAltura(int una_altura){
        return una_altura > 50 && una_altura < 250;
    }
public:
    Persona(string el_nombre, int la_edad, int la_altura){
```

```
        SetNombre(el_nombre);
        SetEdad(la_edad);
        SetAltura(la_altura);
    }
    .....
};
int main(){

    // Persona una_persona; <- Error de compilación 😊

    Persona una_persona ("Juan Carlos", 27, 186); // Estado válido 😊
    .....
}
```

Si los parámetros actuales se leen desde un periférico externo, habrá que crear el objeto después de leer dichos datos:

```
int main(){
    string nombre;
    int edad, altura;

    cin >> nombre;
    cin >> edad;
    cin >> altura;

    Persona una_persona (nombre, edad, altura); // Estado válido 😊
    .....
}
```

http://decsai.ugr.es/~carlos/FP/III_persona_constructor.cpp

Ahora bien, ¿qué ocurre si los parámetros actuales no son valores correctos? Pues que tenemos un problema.

```
int main(){  
    Persona una_persona ("JC", -1, 180);    // "JC", 0, 180  
    // una_persona está en un estado inválido  
    .....  
}
```



La solución a este problema pasa por lanzar una excepción dentro del constructor (ver Tema V) pero está fuera de los objetivos del curso.

Los constructores nos permiten ejecutar automáticamente un conjunto de instrucciones, cada vez que se crea un objeto.

En particular, si se les pasa como parámetro actual los valores a asignar a los datos miembro, permiten la creación de un objeto en un estado válido, en el mismo momento de su definición.

Si los parámetros actuales al constructor no fuesen correctos, la solución pasaría por lanzar una excepción (se verá en un tema posterior). Por ahora, supondremos que los datos pasados son correctos.

Ejercicio. Añadimos un constructor a la clase `Fecha` que definimos en la página 360 para obligar a crear el objeto con los datos del día, mes y año.

Fecha	
- int	dia
- int	mes
- int	anio
+ void	EsFechaCorrecta(int el_dia, int el_mes, int el_anio)
+	Fecha (int el_dia, int el_mes, int el_anio)
+ void	SetDiaMesAnio(int el_dia, int el_mes, int el_anio)
+ int	GetDia()
+ int	GetMes()
+ int	GetAnio()
+ string	ToString()

```
class Fecha {  
private:  
    int dia = 0,  
        mes = 0,  
        anio = 0;  
    .....  
public:  
    Fecha(int el_dia, int el_mes, int el_anio){  
        SetDiaMesAnio(el_dia, el_mes, el_anio);  
    }  
    void SetDiaMesAnio(int el_dia, int el_mes, int el_anio){  
        if (EsFechaCorrecta(el_dia, el_mes, el_anio)){  
            dia = el_dia;  
            mes = el_mes;  
            anio = el_anio;  
        }  
    }  
    .....  
};  
int main(){
```

```
// Fecha nacimiento_JC; // Error de compilación
Fecha nacimiento_JC (27, 2, 1967);    // 27, 2, 1967
Fecha nacimiento_Nadal (3, 6, 1986); // 3, 6, 1986
Fecha otra_fecha (-27, 2, 1967);      // 0, 0, 0
```

Ejercicio. Añadid un constructor a la cuenta bancaria pasándole como parámetros el identificador de cuenta y el saldo inicial.

```
class CuentaBancaria{
private:
    double saldo          = 0.0;
    string identificador; // C++ inicializa los string a ""
    .....
    void SetSaldo (double saldo_propuesto){
        .....
    }
public:
    CuentaBancaria(string identificador_cuenta, double saldo_inicial){
        SetSaldo(saldo_inicial);
        SetIdentificador(identificador_cuenta);
    }
public:
    void SetIdentificador(string identificador_cuenta){
        .....
    }
    .....
};

int main(){
    // CuentaBancaria cuenta; <- Error de compilación

    CuentaBancaria cuenta("20310381450100007510", 3000);
    // cuenta: "2...0", 3000
    // Estado válido

    CuentaBancaria otra_cuenta("ABC", 3000);
    // otra_cuenta: "", 3000
    // Estado inválido
```

Ahora que podemos dar el valor inicial al identificador dentro del constructor, sería lógico imponer que, posteriormente, éste no se pudiese cambiar. Para conseguirlo, bastaría con que el método `SetIdentificador` no fuese público. Nos quedaría:

CuentaBancaria	
- double	saldo
- double	identificador
- bool	EsCorrectoSaldo(double saldo_propuesto)
- bool	EsCorrectoIdentificador(string identificador_propuesto)
- void	SetSaldo(double saldo_propuesto)
+ void	SetIdentificador(string identificador_cuenta)
<hr/>	
+	CuentaBancaria(string identificador_cuenta, double saldo_inicial)
+	string Identificador()
+	double Saldo()
+	void Ingresa(double cantidad)
+	void Retira(double cantidad)
+	void AplicaInteresPorcentual(int tanto_porcento)

```
int main(){
    CuentaBancaria cuenta("20310381450100007510", 3000);
        // cuenta: "2...0", 3000
        // Estado válido

    CuentaBancaria otra_cuenta("20310381450100007511", 500);

    // Las siguientes sentencias darían un error de compilación:
    // cuenta.SetIdentificador("20310381450100009876");
    // otra_cuenta.SetIdentificador("20310381450100003144");
```

III.2.5.2. Sobrecarga de constructores

Se puede proporcionar más de un constructor, siempre que cambien en el tipo o en el número de parámetros. El compilador creará el objeto llamando al constructor correspondiente según corresponda con los parámetros actuales.

Ejemplo. Sobre la cuenta bancaria, proporcionamos un constructor que obligue a pasar el identificador pero no el saldo (en cuyo caso se quedará con cero)

CuentaBancaria	
- double	saldo
- double	identificador
- bool	EsCorrectoSaldo(double saldo_propuesto)
- bool	EsCorrectoIdentificador(string identificador_propuesto)
- void	SetSaldo(double saldo_propuesto)
+ void	SetIdentificador(string identificador_cuenta)
<hr/>	
+	CuentaBancaria(string identificador_cuenta, double saldo_inicial)
+	CuentaBancaria(string identificador_cuenta)
+	string Identificador()
+	double Saldo()
+	void Ingresa(double cantidad)
+	void Retira(double cantidad)
+	void AplicaInteresPorcentual(int tanto_porcento)

```
class CuentaBancaria{
private:
    double saldo          = 0.0;
    string identificador; // Los string se inicializan por defecto a ""
    .....
public:
    CuentaBancaria(string identificador_cuenta, double saldo_inicial){ // 1
        SetIdentificador(identificador_cuenta);
        SetSaldo(saldo_inicial);
    }
    CuentaBancaria(string identificador_cuenta){                      // 2
        SetIdentificador(identificador_cuenta);
        SetSaldo(0.0); // 0 podría dejarse el valor por defecto,
                       // que también es 0.0
    }
    .....
};

int main(){
    CuentaBancaria cuenta1("20310381450100007510", 3000); // 1
        // cuenta1: "2...0", 3000
    CuentaBancaria cuenta2("20310381450100007511"); // 2
        // cuenta2: "2...1", 0.0
```

III.2.5.3. Llamadas entre constructores

Si observamos el ejemplo anterior, podemos apreciar que hay cierto código repetido en los constructores. Lo resolvemos llamando a un constructor desde el otro constructor. El compilador debe garantizar que esta llamada se realiza antes que cualquier otra instrucción del constructor. Para ello, se usa la *lista de inicialización del constructor (constructor initialization list)*. Esta lista se indica con dos puntos justo antes de la llave abierta del constructor:

```
class MiClase{
public:
    MiClase(int un_entero, int otro_entero)
    {
        .....
    }
    MiClase(int entero)
        :MiClase(entero, 5)
    {
        .....
    }
};
```

Ejercicio. Llamad a un constructor dentro del otro en el ejemplo de la cuenta bancaria.

```
class CuentaBancaria{
private:
    double saldo          = 0.0;
    string identificador; // C++ inicializa los string a ""
    .....
public:
    CuentaBancaria(string identificador_cuenta, double saldo_inicial){ // 1
        SetIdentificador(identificador_cuenta);
        SetSaldo(saldo_inicial);
    }
    CuentaBancaria(string identificador_cuenta)                        // 2
        :CuentaBancaria(identificador_cuenta, 0.0)
    {
    }
    .....
};

int main(){
    CuentaBancaria cuenta1("20310381450100007510", 3000);           // 1
        // cuenta1: "2...0", 3000
    CuentaBancaria cuenta2("20310381450100007511");                  // 2
        // cuenta2: "2...1", 0.0
```

III.2.5.4. Constructores sin parámetros

Si se desea, se puede proporcionar un constructor sin parámetros. Hay que indicar que, internamente, C++ proporciona un constructor sin parámetros *oculto* que permite crear el objeto y poco más.

```
class MiClaseSinConstructor{
    .....
};

int main(){
    MiClaseSinConstructor objeto; // Constructor oculto de C++
}
```

Ahora bien, si el programador define otro constructor sin parámetros (en general, cualquier otro constructor), será éste el que se invoque:

```
class MiClase{
    .....
public:
    MiClase(){
        .....
    }
};

int main(){
    MiClase objeto; // Constructor definido por el programador
}
```

Ejemplo. Sobre la clase `Fecha`, añadimos un constructor sin parámetros para que cree una fecha con los datos de la fecha actual.

Fecha	
- int	dia
- int	mes
- int	anio
+ void	EsFechaCorrecta(int el_dia, int el_mes, int el_anio)
+	Fecha()
+	Fecha(int el_dia, int el_mes, int el_anio)
+ void	SetDiaMesAnio (int el_dia, int el_mes, int el_anio)
+ int	GetDia()
+ int	GetMes()
+ int	GetAnio()
+ string	ToString()

Debemos usar la biblioteca `ctime`. No hay que comprender el código del constructor, sino entender cuándo se ejecuta.

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

class Fecha {
private:
    int dia, mes, anio;
    .....
public:
    Fecha(){
        time_t hoy_time_t;
        struct tm * hoy_struct;

        hoy_time_t = time(NULL);
        hoy_struct = localtime ( &hoy_time_t );

        dia  = hoy_struct->tm_mday;
        mes  = hoy_struct->tm_mon + 1;
        anio = hoy_struct->tm_year + 1900 +1;
    }
    Fecha(int el_dia, int el_mes, int el_anio){
        SetDiaMesAnio(el_dia, el_mes, el_anio);
    }
    .....
};

int main(){
    Fecha fecha_hoy; // Constructor sin parámetros
    Fecha nacimiento_JC (27, 2, 1967); // Constructor con parámetros

    cout << fecha_hoy.ToString();
```

http://decsai.ugr.es/~carlos/FP/III_fecha_constructor_sin_parametros.cpp

Ejemplo. Creamos una clase para generar números aleatorios enteros entre \min y \max . Estos valores se pasan como parámetros al constructor. Definid también un constructor sin parámetros, en cuyo caso, se generarán únicamente 0 y 1.

GeneradorAleatorioEnteros	
-
+	GeneradorAleatorioEnteros()
+	GeneradorAleatorioEnteros(int min, int max)
+	double Siguiente()

```
int main(){
    GeneradorAleatorioEnteros generador_numeros_primitiva(1, 49);
    GeneradorAleatorioEnteros aleatorio_0_1;

    cout << "Generador aleatorio de números enteros:\n\n";
    cout << "0 o 1:\n";

    for (int i=0; i<100; i++)
        cout << aleatorio_0_1.Siguiente() << " " ;

    cout << "\n\nCombinación aleatoria de la primitiva:\n";

    for (int i=0; i<6; i++)
        cout << generador_numeros_primitiva.Siguiente() << " " ;
```

```
#include <random> // para la generación de números pseudoaleatorios
#include <chrono> // para la semilla
#include <iostream>
using namespace std;

class GeneradorAleatorioEnteros{
private:
    mt19937 generador_mersenne; // Mersenne twister
    uniform_int_distribution<int> distribucion_uniforme;
public:
    GeneradorAleatorioEnteros()
        :GeneradorAleatorioEnteros(0, 1){
    }
    GeneradorAleatorioEnteros(int min, int max){
        auto semilla =
            chrono::high_resolution_clock::now().time_since_epoch().count();
        generador_mersenne.seed(semilla);
        distribucion_uniforme = uniform_int_distribution<int> (min, max);
    }
    double Siguiente(){
        return distribucion_uniforme(generador_mersenne);
    }
};

int main(){
    .....
}
```

http://decsai.ugr.es/~carlos/FP/III_generador_aleatorio.cpp

III.2.6. Programando como profesionales

Conceptos fundamentales vistos hasta ahora en la construcción de una clase:

- ▷ Ámbitos `private` y `public`.
- ▷ Encapsulación de datos y métodos dentro de una clase.

Ahora bien,

- ▷ ¿Qué incluimos como dato miembro de una clase y qué pasamos como parámetros a sus métodos?
- ▷ ¿Qué preferimos: pocas clases que hagan muchas cosas distintas o más clases que hagan cosas concretas?

III.2.6.1. Datos miembro vs parámetros de los métodos

Los datos miembro se comportan como datos *globales* dentro del objeto, por lo que son directamente accesibles desde cualquier método definido dentro de la clase y no hay que pasarlos como parámetros a dichos métodos.

Por tanto, ¿los métodos de las clases en PDO suelen tener menos parámetros que las funciones *globales* usadas en Programación Procedural?

¡Sí, por supuesto!

Ejercicio. Retomamos el ejemplo de la página 300. Habíamos definido estas funciones:

```
bool EsMayorEdad (int edad){
    return edad >= 18;
}

bool EsAlta (int altura, int edad){
    if (EsMayorEdad(edad))
        return altura >= 190;
    else
        return altura >= 175;
}

int main(){
    .....
    es_mayor_edad = EsMayorEdad(48);
    es_alta       = EsAlta(186, 48);
}
```

¿Cómo quedaría si trabajásemos con una clase?

Bastaría añadir los siguientes métodos a la clase vista en la página 366

```
class Persona{
private:
    string nombre;
    int edad;
    int altura;
    .....
public:
    .....
    bool EsMayorEdad (){
        return edad >= 18;
    }
    bool EsAlta (){
        if (EsMayorEdad())
            return altura >= 190;
        else
            return altura >= 175;
    }
};

int main(){
    Persona una_persona("JC", 48, 186);
    .....
    es_mayor_edad = una_persona.EsMayorEdad();
    es_alta      = una_persona.EsAlta();
}
```

En cualquier caso, en numerosas ocasiones los métodos necesitarán información adicional que no esté descrita en el estado del objeto, como por ejemplo, la cantidad a ingresar o retirar de una cuenta bancaria, o el número de unidades a trasladar un segmento. Esta información adicional serán parámetros de los correspondientes métodos.

No siempre es fácil determinar cuáles deben ser los datos miembros y cuáles los parámetros.

Ejemplo. ¿Incluimos en la cuenta bancaria las cantidades a ingresar y a retirar como datos miembro?

```
class CuentaBancaria{
private:
    double saldo = 0.0;;
    string identificador;
    double cantidad_a_ingresar;
    double cantidad_a_retirar;
public:
    .....
    void SetCantidadIngresar(double cantidad){
        cantidad_a_ingresar = cantidad;
    }
    void Ingresa(){
        saldo = saldo + cantidad_a_ingresar;
    }
    void Retira(){
        saldo = saldo - cantidad_a_retirar;
    }
    .....
};

int main(){
    CuentaBancaria una_cuenta(40);

    una_cuenta.SetCantidadIngresar(25);
    una_cuenta.Ingresa();
}
```

La cantidad a ingresar/retirar no es un dato miembro. Es una información que se necesita sólo en el momento de ingresar/retirar.



Sólo incluiremos como datos miembro aquellas características esenciales que determinan una entidad.

La información adicional que se necesite para ejecutar un método, se proporcionará a través de los parámetros de dicho método.



Ejemplo. ¿Incluimos la longitud como dato miembro del segmento?

SegmentoDirigido		
-	double	x_1
-	double	y_1
-	double	x_2
-	double	y_2
¿-	double	longitud?
.....		
+	double	Longitud()

No debemos incluir `longitud` como un dato miembro. Es una propiedad que puede calcularse en función de las coordenadas.

Las propiedades que se puedan calcular a partir de los datos miembro, se obtendrán a través de un método.

III.2.6.2. Principio de Responsabilidad Única y cohesión de una clase

¿Qué preferimos: pocas clases que hagan muchas cosas distintas o más clases que hagan cosas concretas? Por supuesto, las segundas.

Principio en Programación Dirigida a Objetos.
Principio de Responsabilidad Única
(Single Responsibility Principle)

Un objeto debería tener una única responsabilidad, la cual debe estar completamente encapsulada en la definición de su clase.



A cada clase le asignaremos una única responsabilidad. Esto facilitará:

- ▷ La reutilización en otros contextos
- ▷ La modificación independiente de cada clase (o al menos paquetes de clases)

Heurísticas:

- ▷ Si para describir el propósito de la clase se usan más de 20 palabras, puede que tenga más de una responsabilidad.
- ▷ Si para describir el propósito de una clase se usan las conjunciones y u o, seguramente tiene más de una responsabilidad.

Ejemplo. ¿Es correcto este diseño?:

```
class CuentaBancaria{
private:
    double saldo;
    string identificador;
public:
    CuentaBancaria(double saldo_inicial){
        saldo = saldo_inicial;
    }
    string Identificador(){
        return identificador;
    }
    void SetIdentificador(string identificador_cuenta){
        identificador = identificador_cuenta;
    }
    double Saldo(){
        return saldo;
    }
    void ImprimeSaldo(){
        cout << "\nSaldo = " << saldo;
    }
    void ImprimeIdentificador(){
        cout << "\nIdentificador = " << identificador;
    }
};
```



La responsabilidad de esta clase es gestionar los movimientos de una cuenta bancaria **Y** comunicarse con el dispositivo de salida por defecto. Esto viola el principio de única responsabilidad. Esta clase no podrá reutilizarse en un entorno de ventanas, en el que `cout` no funciona.

Mezclar E/S y cálculos en una misma clase viola el principio de responsabilidad única. Lamentablemente, esto se hace con mucha frecuencia, incluso en multitud de libros de texto.

Jamás mezclaremos en una misma clase métodos de *cálculo* con métodos que accedan a los dispositivos de entrada/salida.

IMPORTANT

Ejemplo. Retomamos el ejemplo de la fecha:

¿No hubiera sido más cómodo sustituir el método ToString por Imprimir?:

```
class Fecha {
private:
    int dia, mes, anio;
    .....
public:
    .....
    void Imprimir(){
        cout << string(dia) + "/" + to_string(mes)
            + "/" + to_string(anio);
    }
};

int main(){
    Fecha nacimiento_JC (27,2,1987);
    nacimiento_JC.Imprimir();
}
```



De nuevo estaríamos violando el principio de responsabilidad única. Debemos eliminar el método Imprimir y usar el que teníamos antes ToString

```
class Fecha {
    .....
    string ToString(){
        return string(dia) + "/" + to_string(mes)
            + "/" + to_string(anio);
    }
};

int main(){
    Fecha nacimiento_JC (27,2,1987);

    cout << nacimiento_JC.ToString();
}
```



Un concepto ligado al principio de responsabilidad única es el de **cohesión (cohesion)**, que mide cómo de relacionadas entre sí están las funciones desempeñadas por un componente software (paquete, módulo, clase o subsistema en general)

Por normal general, cuantos más métodos de una clase utilicen todos los datos miembros de la misma, mayor será la cohesión de la clase.

Ejemplo. ¿Es correcto este diseño?

ClienteCuentaBancaria	
- double	saldo
- string	identificador
- string	nombre
- string	dni
- int	dia_ncmto
- int	mes_ncmto
- int	anio_ncmto

+ ClienteCuentaBancaria(double saldo_inicial, string nombre_cliente, string dni_cliente, int dia_nacimiento, int mes_nacimiento, int anio_nacimiento)	
+ void	SetIdentificador(string identificador_cuenta)
+ string	Identificador()
+ double	Saldo()
+ void	Ingresa(double cantidad)
+ void	Retira(double cantidad)
+ void	AplicaInteresPorcentual(int tanto_por_ciento)
+ string	Nombre()
+ string	DNI()
+ string	FechaNacimiento()

Puede apreciarse que hay **grupos** de métodos que acceden a otros **grupos** de datos miembro. Esto nos indica que pueden haber varias responsabilidades mezcladas en la clase anterior.

Solución: Trabajar con dos clases: CuentaBancaria y Cliente:

Cliente	
-	string nombre
-	string dni
-	int dia_ncmto
-	int mes_ncmto
-	int anio_ncmto
+	Cliente(string nombre_cliente, string dni_cliente, int dia_nacimiento int mes_nacimiento, int anio_nacimiento)
+	string Nombre()
+	string DNI()
+	string FechaNacimiento()

Si necesitamos conectar ambas clases, podemos crear una tercera clase que contenga un Cliente y una colección de cuentas asociadas. Esto se verá en el último tema.

III.2.6.3. Funciones vs Clases

¿Cuándo usaremos funciones y cuándo clases?

- ▷ Como norma general, usaremos clases.
- ▷ Usaremos funciones para implementar tareas muy genéricas que operan sobre tipos de datos simples y que preveamos se pueden utilizar en programas distintos.

```
bool  SonIguales(double uno, double otro)
bool  EsPrimo(int dato)
char  ToMayuscula(char letra)
double Redondea(double numero)
.....
```

Ejemplo. Retomamos el ejemplo del segmento de la página 361. El método privado `SonCorrectas` comprobaba que las coordenadas del punto inicial fuese distintas del punto final:

```
bool SonCorrectas(double abs_1, ord_1, abs_2, ord_2){
    return !(abs_1 == abs_2 && ord_1 == ord_2);
}
```

Al estar trabajando con reales, hubiese sido mejor realizar la comparación con un margen de error:

```
bool SonCorrectas(double abs_1, ord_1, abs_2, ord_2){
    return !(SonIguales(abs_1,abs_2) && SonIguales(ord_1,ord_2));
}
```

¿Dónde definimos `SonIguales`? No es un método relativo a un objeto `Segmento`, por lo que no es un método público. Podría ser:

- ▷ O bien un método privado.
- ▷ O bien una función global.

Si el criterio de igualdad entre reales es específico para la clase (queremos un margen de error específico para las coordenadas de un segmento) usaremos un método privado.

Pero en este ejemplo, es de esperar que otras clases como `Triangulo`, `Cuadrado`, etc, usen el mismo criterio de igualdad entre reales. Por tanto, preferimos una función.

```
bool SonIguales(double uno, double otro) {
    return abs(uno-otro) <= 1e-6;
}

class SegmentoDirigido{
private:
    double x_1 = 0.0,
           y_1 = 0.0,
           x_2 = 0.0,
           y_2 = 0.0;

    bool SonCorrectas(double abs_1, ord_1, abs_2, ord_2){
        return !(SonIguales(abs_1,abs_2) && SonIguales(ord_1,ord_2));
    }
public:
    void SetCoordenadas(double origen_abscisa,
                       double origen_ordenada,
                       double final_abscisa,
                       double final_ordenada){
        if (SonCorrectas(origen_abscisa, origen_ordenada,
                        final_abscisa, final_ordenada)){
            x_1 = origen_abscisa;
            y_1 = origen_ordenada;
            x_2 = final_abscisa;
            y_2 = final_ordenada;
        }
    }
    .....
};

class Triangulo{
    // Aquí también llamamos a SonIguales
    .....
};
```

Ejemplo. Una clase `Fraccion` podría tener el siguiente esquema (los valores los pasamos en el constructor y no se permite su posterior modificación)

Fraccion	
- int	numerador
- int	denominador
+ Fraccion(int el_numerador, int el_denominador)	
+ int	Numerador()
+ int	Denominador()
+ void	Normaliza()
+ double	Division()
+ string	ToString()

```
int main(){
    Fraccion una_fraccion(4, 10);    // 4/10

    una_fraccion.Normaliza();         // 2/5
    cout << una_fraccion.ToString();
    .....
```

Para implementar el método `Normaliza`, debemos dividir numerador y denominador por el máximo común divisor. ¿Dónde lo implementamos? El cómputo del MCD no está únicamente ligado a la clase `Fraccion`, por lo que usamos una función global.

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

int MCD(int primero, int segundo){
    .....
}

class Fraccion{
    .....
    void Normaliza(){
        int mcd;

        mcd = MCD(numerador, denominador);
        numerador = numerador/mcd;
        denominador = denominador/mcd;
    }
};

int main(){
    Fraccion una_fraccion(4, 10);    // 4/10

    una_fraccion.Normaliza();         // 2/5
    cout << una_fraccion.ToString();
    .....
```

http://decsai.ugr.es/~carlos/FP/III_fraccion.cpp

Como norma general, usaremos clases para modularizar los programas. En ocasiones, también usaremos funciones para implementar tareas muy genéricas que operan sobre tipos de datos simples.

III.2.6.4. Comprobación y tratamiento de las precondiciones

En la página 312 se vieron las precondiciones de una función. La misma definición se aplica sobre los métodos. Ahora vamos a responder las siguientes preguntas:

- ▷ ¿Debemos comprobar dentro de la función/método si se satisfacen sus precondiciones?
- ▷ Si decidimos comprobarlo, ¿qué hacemos en el caso de que se violen?

Ejemplo. Función factorial.

Versión en la que no se comprueba la precondición:

```
// Prec: 0 <= n <= 20
long long Factorial (int n){
    int i;
    long long fact = 1;

    for (i = 2; i <= n; i++)
        fact = fact * i;

    return fact;
}
```

Versión en la que se comprueba la precondición:

```
long long Factorial (int n){
    int i;
    long long fact = 1;

    if (n >= 0 && n <= 20){
        for (i = 2; i <= n; i++)
```

```
        fact = fact * i;
    }
    return fact;
}
```

En este ejemplo, podemos optar por omitir la comprobación. Si se viola, el resultado es un desbordamiento aritmético y el factorial contendrá un valor indeterminado pero no supone un error grave del programa.

Ejemplo. Clase CuentaBancaria.

Versión en la que no se comprueba la precondición:

```
class CuentaBancaria{
    .....
    // Prec: cantidad > 0
    void Ingresa(double cantidad){
        saldo = saldo + cantidad;
    }
};
```



Versión en la que se comprueba la precondición:

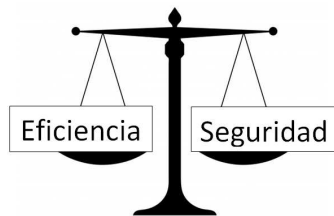
```
class CuentaBancaria{
    .....
    void Ingresa(double cantidad){
        if (cantidad > 0)
            saldo = saldo + cantidad;
    }
};
```

En este caso, es mejor comprobar la precondición ya que no podemos permitir un saldo negativo.

¿Cuándo debemos comprobar las precondiciones?

*Si la violación de una precondición de una función o un método **público** puede provocar errores de ejecución graves, dicha precondición debe comprobarse dentro del método.*

En otro caso, puede omitirse (sobre todo si prima la eficiencia)



¿Comprobar Precondiciones dentro del método/función?

¿Y qué hacemos si comprobamos las precondiciones de una función/método y no se satisfacen? Posibles soluciones:

▷ **No hacer nada.**

Si la cantidad a ingresar en una cuenta bancaria es negativa, no lo permitimos y dejamos la cantidad que hubiese.

```
class CuentaBancaria{
    .....
    void Ingresa(double cantidad){
        if (cantidad > 0)
            saldo = saldo + cantidad;
        // no hay else
    }
};
```

▷ **Realizar una acción por defecto.**

Si la cantidad a ingresar es negativa, la convertimos a positivo y la ingresamos.

```
class CuentaBancaria{
    .....
    void Ingresa(double cantidad){
        if (cantidad > 0)
            saldo = saldo + cantidad;
        else
            saldo = saldo - cantidad;
    }
};
```

La elección de una alternativa u otra dependerá del problema concreto en cada caso, pero lo usual será no hacer nada. En el ejemplo anterior, el hecho de que la cantidad pasada como parámetro sea negativa, nos está indicando algún problema y muy posiblemente estaremos cometiendo un error al ingresar el valor en positivo.

Esta misma situación se presenta con los constructores. Ya vimos en la página 373 que si los parámetros actuales al constructor son incorrectos, lo único que podíamos hacer es dejar asignados los indicados en la inicialización de los datos miembro.

```
class CuentaBancaria{
private:
    double saldo          = 0.0;
    string identificador; // Los string se inicializan por defecto a ""
    .....
public:
    CuentaBancaria(string identificador_cuenta, double saldo_inicial){
        SetSaldo(saldo_inicial);
        SetIdentificador(identificador_cuenta);
    }
    .....
};

int main(){
    CuentaBancaria cuenta(":-(", -3000); // identificador = "",saldo = 0.0
```

¿Y cómo sabemos que se ha violado la precondition desde el sitio en el que se invoca al método/función?

```
class CuentaBancaria{
    .....
    void Ingresa(double cantidad){
        if (cantidad > 0)
            saldo = saldo + cantidad;
        else
            cout << "La cantidad a ingresar no es positiva";
    }
};
```



Nunca haremos este tipo de notificaciones. Ningún método de la clase CuentaBancaria debe acceder al periférico de entrada o salida.

Soluciones para notificar el error:

- ▷ Notificarlo *a la antigua usanza* devolviendo un código de error
- ▷ Mucho mejor: devolver una excepción

En el tema V veremos las excepciones. Veamos ahora la notificación devolviendo un código de error:

- ▷ Si trabajamos con un método/función `void`, lo convertimos en un método/función que devuelva un código de error. El tipo devuelto será `bool` si sólo notificamos la existencia o no del error. Si hay varios posibles errores, lo mejor es usar un enumerado.
- ▷ Si el método/función no es un `void`, podemos usar el mismo valor devuelto para albergar el código de error, pero éste no debe ser uno de los valores *legales* de los devueltos por el método/función.
- ▷ Ninguna de las anteriores soluciones es aplicable a los constructores ya que éstos no pueden devolver ningún valor.

Ejemplo. En la cuenta bancaria transformamos el método `Ingresar` para que devuelva el código de error:

```
void Ingresar(double cantidad) --> bool Ingresar(double cantidad)

class CuentaBancaria{
.....
    bool Ingresar(double cantidad){
        bool error;

        error = (cantidad < 0);

        if (!error)
            saldo = saldo + cantidad;

        return error;
    }
};

int main(){
.....
    bool error_cuenta;
    error_cuenta = cuenta.Ingresar(dinero);

    if (error_cuenta)
        cout << "La cantidad a ingresar no es positiva";
```

En cualquier caso, si devolvemos un código de error, en el sitio de la llamada, tendremos que poner siempre el `if` correspondiente, lo que resulta bastante tedioso.

La mejor solución pasa por devolver una excepción (ver tema V) dentro del método/función en el que se viola la precondition. Esto también es válido para el constructor.

Si decidimos que es necesario **comprobar** alguna precondition dentro de un método/función/constructor y ésta se viola, ¿qué debemos hacer como respuesta?

- ▷ O bien no hacemos nada (salvo lanzar una excepción, tal y como se indica en el siguiente párrafo) o bien realizamos una acción por defecto. La elección apropiada depende de cada problema, aunque lo usual será no hacer nada.
- ▷ Además de lo anterior, si decidimos que es necesario **notificar** dicho error al cliente que ejecuta el método/función podemos devolver un código de error o, mucho mejor, lanzar una excepción (ver tema V). En el caso del constructor, sólo podemos lanzar una excepción.

La notificación jamás se hará imprimiendo un mensaje en pantalla desde dentro del método/función.

III.2.7. Registros

En algunas ocasiones, la clase que queremos construir es tan simple que no tiene métodos (comportamiento) asociados, ni restricciones sobre los valores asignables. Únicamente sirve para agrupar algunos datos. Para estos casos, usamos mejor un tipo específico proporcionado por C++ (también habitual en los lenguajes de programación): el tipo `struct`

Un *registro* o `struct` permite almacenar varios elementos de (posiblemente) distintos tipos y gestionarlos como uno sólo. Cada uno de los datos agrupados en un `struct` es un *campo*.

Los campos de un `struct` se suponen relacionados, que hacen referencia a una misma entidad. Cada campo tiene un *nombre*.

Un ejemplo típico es la representación en memoria de un *punto* en un espacio bidimensional. Un punto está caracterizado por dos valores: *abscisa* y *ordenada*. En este caso, tanto *abscisa* como *ordenada* son del mismo tipo, supongamos que sea `int`.

```
struct Punto2D {  
    int abscisa;  
    int ordenada;  
};
```

Se ha definido un tipo de dato registro llamado `Punto2D`. Los datos de este tipo están formados por dos campos llamados `abscisa` y `ordenada`, ambos de tipo `int`.

Cuando se declara un dato de este tipo:

```
Punto2D un_punto;
```

se crea una variable llamada `punto`, y a través de su nombre se puede acceder a los campos que la conforman mediante el operador punto (`.`).

Por ejemplo, podemos establecer los valores de la *abscisa* y *ordenada* de `un_punto` a 4 y 6 respectivamente con las instrucciones:

```
un_punto.abscisa = 4;  
un_punto.ordenada = 6;
```

Es posible declarar variables `struct` junto a la definición del tipo (aunque normalmente lo haremos de forma separada):

```
struct Punto2D {  
    int abscisa;  
    int ordenada;  
} un_punto, otro_punto;
```

Un ejemplo de `struct` heterogéneo:

```
struct Persona {  
    string nombre;  
    string apellidos;  
    string NIF;  
    char categoria;  
    double salario_bruto;  
};
```

No se supone ningún orden establecido entre los campos de un `struct` y no se impone ningún orden de acceso a éstos (por ejemplo, puede inicializarse el tercer campo antes del primero).

No puede leerse/escribirse un `struct`, sino a través de cada uno de sus campos, separadamente. Por ejemplo, para leer los valores de `un_punto` desde la entrada estándar:

```
cin >> un_punto.abscisa;  
cin >> un_punto.ordenada;
```

y para escribirlos en la salida estándar:

```
cout << "(" << un_punto.abscisa << ", " << un_punto.ordenada << ");
```

Es posible asignar un `struct` a otro y en consecuencia, un `struct` puede aparecer tanto como *lvalue* y *rvalue* en una asignación.

```
otro_punto = un_punto;
```

Los campos de un `struct` pueden emplearse en cualquier expresión:

```
dist_x = abs(un_punto.abscisa - otro_punto.abscisa);
```

siendo, en este ejemplo, `un_punto.abscisa` y `otro_punto.abscisa` de tipo `int`.

III.2.7.1. `struct` y funciones

Un dato `struct` puede pasarse como parámetro a una función y una función puede devolver un `struct`.

Por ejemplo, la siguiente función recibe dos `struct` y devuelve otro.

```
Punto2D CalculaPuntoMedio (Punto2D punto1, Punto2D punto2)
{
    Punto2D punto_medio;

    punto_medio.abscisa = (punto1.abscisa + punto2.abscisa) / 2;
    punto_medio.ordenada = (punto1.ordenada + punto2.ordenada) / 2;

    return (punto_medio);
}
```

III.2.7.2. Ámbito de un `struct`

Una variable `struct` es una variable más y su ámbito es de cualquier variable:

- ▷ Puede ser una variable global, aunque ya sabe que el uso de esta clase de variables no está permitido en esta asignatura.
- ▷ Como cualquier variable local a una función, puede usarse desde su declaración hasta el fin de la función en que ha sido declarada. Por ejemplo, en la función `CalculaPuntoMedio` la variable `punto_medio` es una variable local y se comporta como cualquier otra variable local, independientemente de que sea un `struct`.
- ▷ El ámbito más reducido es el nivel de bloque: si el `struct` se declara dentro de un bloque sólo podría usarse dentro de él.

La variable `punto_medio` es una variable local de la función `CalculaPuntoMedio` y se comporta como cualquier otra variable local, independientemente de que sea un `struct`. Lo mismo podría decirse si el `struct` se declarara dentro de un bloque: sólo podría usarse dentro de ese bloque.

```
...
while (sigo) {
    Punto2D punto_aux;

    // punto_aux solo es accesible solo dentro del ciclo while.
    // Se crea un struct nuevo en cada iteración y "desaparece"
    // el de la anterior.

    ...
}
```

III.2.7.3. Inicialización de los campos de un struct

Un `struct` puede inicializarse en el momento de su declaración. El objetivo es asignar un valor inicial a sus campos evitando que pueda usarse con valores basura.

El procedimiento es muy simple, y debe tenerse en cuenta el orden en que fueron declarados los campos del `struct`. Los valores iniciales se especifican entre llaves, separados por comas.

Por ejemplo, para inicializar un `struct Persona` podríamos escribir:

```
Persona una_persona = {"", "", "", 'A', 0.0};
```

que asigna los valores *cadena vacía* a los campos de tipo `string` (nombre, apellidos y NIF), el carácter `A` al campo `categoria` y el valor `0.0` al campo `salario_bruto`.

III.2.8. Datos miembro constantes

Recuperamos el ejemplo de la cuenta bancaria. Recordad que el método `SetIdentificador` era privado, por lo que, una vez asignado un identificador en el constructor, éste no podía cambiarse.

```
int main(){
    CuentaBancaria cuenta("20310381450100007510", 3000);

    // La siguiente sentencia da error de compilación
    // El método SetIdentificador es private:

    cuenta.SetIdentificador("20310381450100007511");
```

Desde fuera, hemos conseguido que el identificador sea constante. Pero ahora vamos a obligar a que también sea constante dentro de la clase.

Vamos a definir datos miembros constantes, es decir, que se producirá un error en tiempo de compilación si incluimos una sentencia en algún método de la clase que pretenda modificarlos.

Tipos de dato miembro constantes:

- ▷ **Constantes a nivel de objeto (*object constants*)** : Cada objeto de la clase tiene su propio valor de constante.
- ▷ **Constantes estáticas (*static constants*) o constantes a nivel de clase (*class constants*)** : Todos los objetos de una clase comparten el mismo valor de constante.

III.2.8.1. Constantes a nivel de objeto

- ▷ Cada objeto de una misma clase tiene su propio valor de constante.
- ▷ Se declaran dentro de la clase anteponiendo `const` al nombre de la constante.
- ▷ El valor a asignar lo recibe como un parámetro más en el constructor. La inicialización debe hacerse en un la lista de inicialización del constructor. Son construcciones del tipo `<dato_miembro> (<valor inicial>)` separadas por coma. Recordad que la lista de inicialización del constructor va antes del paréntesis del constructor, y con dos puntos al inicio.

```
class MiClase{
private:
    const double CTE_REAL;
    const int    CTE_STRING;
public:
    MiClase(double un_real, int un_string)
        :CTE_REAL(un_real) ,
        :CTE_STRING(un_string)
    {
        CTE_REAL = un_real;    // Error de compilación. No es el sitio
    }
    void UnMetodo(){
        CTE_REAL  = 0.0;      // Error de compilación. Es constante
        CTE_STRING = "Si";    // Error de compilación. Es constante
    }
    .....
};

int main(){
    MiClase un_objeto(6.3, "Si");    // CTE_REAL = 6.3, CTE_STRING = "Si"
    MiClase un_objeto(5.8, "No");    // CTE_REAL = 5.8, CTE_STRING = "No"
```

Realmente, en la lista de inicialización se pueden definir (dar un valor inicial) todos los datos miembros, no sólo las constantes a nivel de objeto.

```
class MiClase{
private:
    double dato_miembro;
public:
    .....
    MiClase(double parametro)
        :dato_miembro(parametro)
    { }
    .....
}
```

De hecho, esta es la forma *recomendada* en C++ de inicializar los datos miembro. Por comodidad, nosotros usaremos inicialización del dato miembro en el mismo lugar de la declaración, salvo lógicamente los datos miembros constantes, que es obligatorio hacerlo en la lista de inicialización.

Ejercicio. Recuperad la clase `CuentaBancaria` y definid el identificador como una constante.

```
class CuentaBancaria{
private:
    double saldo;
    const string IDENTIFICADOR;
public:
    CuentaBancaria(string identificador_cuenta, double saldo_inicial)
        :IDENTIFICADOR(identificador_cuenta)
    {
        SetSaldo(saldo_inicial);
    }
    CuentaBancaria(string identificador_cuenta)
        :IDENTIFICADOR(identificador_cuenta)
    {
        SetSaldo(0.0);
    }
    string Identificador(){
        return IDENTIFICADOR;
    }
    .....
};

int main(){
    CuentaBancaria una_cuenta("20310087370100001345", 100);
    .....
```

III.2.8.2. Constantes a nivel de clase

Todos los objetos de una clase comparten el mismo valor de constante.

El valor a asignar se indica en la inicialización del dato miembro.

En UML, las constantes estáticas se resaltan subrayándolas.

Constantes estáticas NO enteras

▷ Se **declaran dentro** de la definición de la clase:

```
class <nombre_clase>{
    .....
    static const <tipo> <nombre_cte>;
    .....
};
```

▷ Se **definen fuera** de la definición de la clase:

```
const <tipo> <nombre_clase>::<nombre_cte> = <valor>;
```

```
class MiClase{
private:
    static const double CTE_REAL_PRIVADA;
    .....
};

const double MiClase::CTE_REAL_PRIVADA = 6.7;
```

```
int main(){
    MiClase un_objeto;
```

:: es el *operador de resolución de ámbito (scope resolution operator)*. Se utiliza, en general, para poder realizar la declaración de un miembro dentro de la clase y su definición fuera. Esto permite la *compilación se-*

parada (separate compilation) . Se verá con más detalle en el segundo cuatrimestre.

Constantes estáticas enteras

▷ O bien como antes:

```
class MiClase{
private:
    static const int CTE_ENTERA_PRIVADA;
    .....
};
const int MiClase::CTE_ENTERA_PRIVADA = 6;
```

▷ O bien se definen en el mismo sitio de la declaración:

```
class MiClase{
private:
    static const int CTE_ENTERA_PRIVADA = 4;
    .....
};
```

En el siguiente tema vamos a necesitar las constantes estáticas enteras para dimensionar vectores dentro de una clase.

Bibliografía recomendada para este tema:

- ▷ A un nivel menor del presentado en las transparencias:
 - Capítulo 3 (para las funciones) y primera parte del capítulo 6 (para las clases) de Deitel & Deitel
- ▷ A un nivel similar al presentado en las transparencias:
 - Capítulos 6 y 13 de Gaddis.
 - Capítulos 4, 5 y 6 de Mercer
 - Capítulo 4 de Garrido (sólo funciones, ya que este libro no incluye nada sobre clases)
- ▷ A un nivel con más detalles:
 - Capítulos 10 y 11 de Stephen Prata.

Índice alfabético

- ámbito (scope), 267, 323
- ámbito público (public scope), 343
- ámbito privado (private scope), 343
- índice de variación, 112
- algoritmo (algorithm), 3
- ascii extendido (extended ascii), 71
- asociatividad (associativity), 48
- biblioteca (library), 13
- bit, 43
- bucle controlado por condición (condition-controlled loop), 186
- bucle controlado por contador (counter controlled loop), 186
- bucle post-test (post-test loop), 186
- bucle pre-test (pre-test loop), 186
- buffer, 87
- byte, 43
- código binario (binary code), 6
- código fuente (source code), 9
- cabecera (header), 254
- cadena de caracteres (string), 77
- clase (class), 320
- codificación (coding), 71
- cohesión (cohesion), 396
- coma flotante (floating point), 51
- compilación separada (separate compilation), 421
- compilador (compiler), 12
- componentes léxicos (tokens), 18
- comportamiento (behaviour), 318, 321
- condición (condition), 101
- constante (constant), 30
- constantes a nivel de clase (class constants), 416
- constantes a nivel de objeto (object constants), 416
- constantes estáticas (static constants), 416
- constructor (constructor), 370
- cursor (cursor), 87
- dato (data), 3, 14
- datos globales (global data), 293
- datos locales (local data), 267
- datos miembro (data member), 321
- declaración (declaration), 14
- declaración de un dato (data declaration), 25
- desbordamiento aritmético (arithmetic overflow), 60
- diagrama de flujo (flowchart), 99
- efectos laterales (side effects), 293
- encapsulación (encapsulation), 321
- entero (integer), 44
- entrada de datos (data input), 15
- enumerado (enumeration), 173
- errores en tiempo de compilación (compilation error), 20
- errores en tiempo de ejecución (execution error), 21
- errores lógicos (logic errors), 21
- espacio de nombres (namespace), 17
- especificador de acceso (access specifier), 323
- estado (state), 321
- estado inválido, 363
- estructura condicional (conditional structure), 101
- estructura condicional doble (else conditional structure), 114
- estructura repetitiva (iteration/loop), 186
- estructura secuencial (sequential control flow structure), 101
- evaluación en ciclo corto (short-circuit evaluation), 185
- evaluación en ciclo largo (eager evaluation), 185
- exponente (exponent), 51
- expresión (expression), 37
- expresiones aritméticas (arithmetic expression), 58
- filtro (filter), 189
- flujo de control (control flow), 19, 98
- función (function), 15, 41
- funciones globales (global functions), 317
- funciones miembro (member functions), 321
- hardware, 2
- identificador (identifier), 14
- implementación de un algoritmo (algorithm implementation), 9
- indeterminación (undefined), 55
- infinito (infinity), 55
- instancia (instance), 320
- interfaz (interface), 348
- iteración (iteration), 187
- l-value, 38
- lógico (boolean), 81
- lectura anticipada, 196
- lenguaje de programación (programming language), 7
- lenguaje ensamblador (assembly language), 7
- lenguajes de alto nivel (high level language), 7
- lista de inicialización del constructor (constructor initialization list), 380
- literal (literal), 29
- literales de cadenas de caracteres (string literals), 29

literales de caracteres (character literals), 29	operador unario (unary operator), 39	programación modular (modular programming), 317
literales enteros (integer literals), 45	parámetros (parameter), 41	programación orientada a objetos (object oriented programming), 318
literales lógicos (boolean literals), 29	parámetros actuales (actual parameters), 255	programación procedural (procedural programming), 317
literales numéricos (numeric literals), 29	parámetros formales (formal parameters), 255	programador (programmer), 2
métodos (methods), 318, 321	paso de parámetro por valor (pass-by-value), 256	r-value, 38
módulo (module), 253	pila (stack), 277	rango (range), 42
mantisa (mantissa), 51	precisión (precision), 53	real (float), 50
marco de pila (stack frame), 277	precondición (precondition), 312	redondeo (rounding), 52
modularización (modularization), 316	prevalencia de nombre (name hiding), 292	reglas sintácticas (syntactic rules), 18
mutuamente excluyente (mutually exclusive), 123	principio de programación - ocultación de información (programming principle - information hiding), 285	salida de datos (data output), 16
notación científica (scientific notation), 50	principio de programación - sencillez (programming principle - simplicity), 182	sentencia (sentence/statement), 14
notación infija (infix notation), 39	principio de programación - una única vez (programming principle - once and only once), 95	sentencia condicional (conditional statement), 102
notación prefija (prefix notation), 39	procedimiento (procedure), 287	sentencia condicional doble (else conditional statement), 114
objeto (object), 320	programa (program), 9	software, 2
objetos (objects), 318	programación (programming), 11	tipos de datos (data types), 14
operador (operator), 15, 41	programación declarativa (declarative programming), 317	tipos de datos primitivos (primitive data types), 25
operador binario (binary operator), 39	programación estructurada (structured programming), 317	transformación de tipo (casting), 60
operador de asignación (assignment operator), 15	programación funcional (functional programming), 317	uml, 344
operador de casting (casting operator), 69		unicode, 71
operador de resolución de ámbito (scope resolution operator), 420		usuario (user), 2
operador n-ario (n-ary operator), 39		valor (value), 15
		variables (variables), 30