



Fundamentos de Programación

Curso 2019/2020

© Copyright: Juan Carlos Cubero. Universidad de Granada.

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

Cubero@decsai.ugr.es



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

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

El color rojo se usará para destacar partes especialmente importantes

Algunos símbolos usados:



Principio de Programación.



denota algo especialmente importante.



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



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



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



denota código que nos da escalofríos de sólo verlo.



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



denota un consejo de programación.



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



Reseña histórica.



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

Contenidos

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

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

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

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

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

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

II.3.2.4. El bucle <code>for</code> como ciclo controlado por condición	285
II.3.3. Otras (perniciosas) estructuras de control	293
III. Vectores y Matrices	306
III.1. Fundamentos	307
III.1.1. Introducción	307
III.1.1.1. Motivación	307
III.1.1.2. Declaración	309
III.1.2. Operaciones básicas	310
III.1.2.1. Acceso	310
III.1.2.2. Asignación	312
III.1.2.3. Lectura y escritura	314
III.1.2.4. Inicialización	315
III.1.3. Trabajando con las componentes	316
III.1.3.1. Representación en memoria	316
III.1.3.2. Gestión de componentes utilizadas	319
III.1.3.3. El tipo <code>string</code> como secuencia de caracteres	329
III.2. Recorridos sobre vectores	331
III.2.1. Algoritmos de búsqueda	331
III.2.1.1. Búsqueda Secuencial	332
III.2.1.2. Búsqueda Binaria	333

III.2.1.3. Otras búsquedas	336
III.2.2. Recorridos que modifican componentes	339
III.2.2.1. Inserción de un valor	339
III.2.2.2. Eliminación de un valor	340
III.2.3. Algoritmos de ordenación	342
III.2.3.1. Ordenación por Selección	344
III.2.3.2. Ordenación por Inserción	348
III.2.3.3. Ordenación por Intercambio Directo (Método de la Burbuja)	351
III.3. Matrices	357
III.3.1. Declaración y operaciones con matrices	357
III.3.1.1. Declaración	357
III.3.1.2. Acceso y asignación	359
III.3.1.3. Inicialización	360
III.3.1.4. Representación en memoria (Ampliación)	361
III.3.1.5. Más de dos dimensiones	362
III.3.2. Gestión de componentes útiles con matrices	363
III.3.2.1. Se usan todas las componentes	363
III.3.2.2. Se ocupan todas las columnas pero no todas las filas (o al revés)	368
III.3.2.3. Se ocupa un bloque rectangular	370

III.3.2.4. Se ocupan las primeras filas, pero con tamaños distintos	375
IV. Funciones y Clases	379
IV.1. Funciones	380
IV.1.1. Fundamentos	380
IV.1.1.1. Las funciones realizan una tarea	380
IV.1.1.2. Definición	381
IV.1.1.3. Parámetros formales y actuales	383
IV.1.1.4. Datos locales	394
IV.1.1.5. Precondiciones	404
IV.1.1.6. Documentación de una función	405
IV.1.1.7. La Pila	408
IV.1.1.8. Funciones void	413
IV.1.1.9. El principio de ocultación de información . .	417
IV.1.2. Diseño de funciones	421
IV.1.2.1. Funciones genéricas vs funciones específicas	421
IV.1.2.2. ¿Cuántos parámetros pasamos?	423
IV.1.2.3. ¿Qué parámetros pasamos?	432
IV.1.3. Cuestiones varias	439
IV.1.3.1. Ámbito de un dato (revisión). Variables globales	439
IV.1.3.2. Cuestión de estilo	443

IV.1.3.3. Sobrecarga de funciones (Ampliación) . . .	446
IV.2. Clases	448
IV.2.1. Motivación. Clases y objetos	449
IV.2.2. Encapsulación	453
IV.2.2.1. Datos miembro	455
IV.2.2.2. Métodos	460
IV.2.2.3. Llamadas entre métodos dentro del propio objeto	467
IV.2.2.4. Controlando el acceso a los datos miembro	471
IV.2.3. Ocultación de información	474
IV.2.3.1. Ámbito público y privado	474
IV.2.3.2. UML: Unified modeling language	477
IV.2.3.3. Métodos Get y Set: protegiendo los datos miembro	478
IV.2.3.4. Datos miembro vs parámetros de los métodos	489
IV.2.3.5. Comprobación de precondiciones en los métodos Set	495
IV.2.3.6. Métodos privados	503
IV.2.3.7. Ámbito de un objeto	510
IV.2.4. Constructores	512
IV.2.4.1. Definición de constructores	512
IV.2.4.2. Constructores sin parámetros	520

IV.2.4.3. Sobrecarga de constructores	527
IV.2.4.4. Llamadas entre constructores	530
IV.2.4.5. Estado inválido de un objeto	531
IV.2.5. Comprobación y tratamiento de las precondiciones .	538
IV.2.5.1. Comprobar o no comprobar	538
IV.2.5.2. Tratamiento de la violación de la precondición	543
IV.2.5.3. Comprobación de las precondiciones en el constructor	548
IV.2.6. Registros (structs)	556
IV.2.6.1. El tipo de dato struct	556
IV.2.6.2. Funciones/métodos y el tipo struct	558
IV.2.6.3. Ámbito de un struct	559
IV.2.6.4. Inicialización de los campos de un struct . .	560
IV.2.7. Datos miembro constantes	561
IV.2.7.1. Constantes a nivel de objeto	562
IV.2.7.2. Constantes a nivel de clase (estáticas) . . .	565
IV.2.8. Vectores y objetos	567
IV.2.8.1. Los vectores como datos locales de un mé- todo	568
IV.2.8.2. Los vectores como datos miembro	570
IV.2.8.3. Los vectores como parámetros a un método	580

IV.2.8.4. Comprobación de las precondiciones trabajando con vectores	581
IV.2.9. La clase Secuencia de caracteres	584
IV.2.9.1. Métodos básicos	584
IV.2.9.2. Métodos de búsqueda	591
IV.2.9.3. Algoritmos de ordenación	609
IV.2.10. La clase string	616
IV.2.10.1. Métodos básicos	616
IV.2.10.2. El método ToString	620
IV.2.10.3. Lectura de un string con getline	621
IV.2.11. Diseño de una clase	622
IV.2.11.1. Datos miembro y parámetros	622
IV.2.11.2. Principio de Responsabilidad Única y cohesión de una clase	627
IV.2.11.3. Separación de Entradas/Salidas y Cómputos	631
IV.2.11.4. Tareas primitivas	634
IV.2.11.5. Funciones vs Clases	636

Tema I

Introducción a la Programación

Objetivos:

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

I.1. El ordenador, algoritmos y programas

I.1.1. El Ordenador: Conceptos Básicos

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



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

I.1.2. Datos y Algoritmos

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

► **Características básicas:**

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

► **Características esenciales:**

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

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

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

Ejemplo. Algoritmo de la media aritmética de N valores.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

...

Instrucciones no válidas en un algoritmo:

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

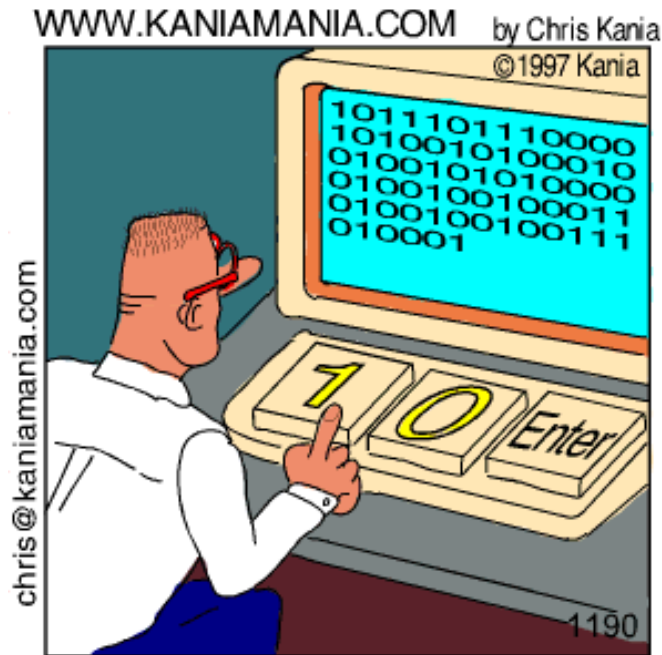
Una vez diseñado el algoritmo, debemos escribir las órdenes que lo constituyen en un lenguaje que entienda el ordenador.

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



I.1.3. Lenguajes de programación

Código binario (Binary code) :



Real programmers code in binary.

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



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

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

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

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

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

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

Reseña histórica del lenguaje C++:

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

¿Qué programas se han hecho en C++?

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

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

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

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

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

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

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

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

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

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

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

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

Pitagoras.cpp

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

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

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

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

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

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

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

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

Un programa incluirá la implementación de uno o más algoritmos.

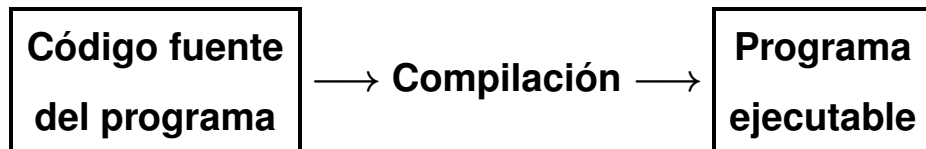
Ejemplo. Programa para dibujar planos de pisos.

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

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

I.1.4. Compilación

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



La extensión en Windows de los programas ejecutables es `.exe`

Código Fuente: Pitagoras.cpp

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

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



Programa Ejecutable: Pitagoras.exe

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

I.2. Especificación de programas

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

I.2.1. Organización de un programa

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

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

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

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

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

También aparece

```
using namespace std;
```

La finalidad de esta declaración se verá posteriormente.

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

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

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

Veamos algunos tipos de sentencias usuales:

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

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

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

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

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

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

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

▷ **Sentencias de asignación**

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

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

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

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

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

▷ **Sentencias de entrada de datos**

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

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

```
cin >> lado1;
```

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

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

▷ **Sentencias de salida de datos**

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

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

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

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

Mostraría en pantalla lo siguiente:

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

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

```
cout << 3.1415927;
```

- Si ponemos un dato, se imprime su contenido.

```
cout << hipotenusa;
```

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

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

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

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

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

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

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

I.2.2. Elementos básicos de un lenguaje de programación

I.2.2.1. Tokens y reglas sintácticas

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

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

58 main ; (== = hipotenusa * /*

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

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

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

I.2.2.2. Palabras reservadas

Suelen ser tokens formados por caracteres alfabéticos.

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

Algunos usos:

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

Estos se verán en el tema II.

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

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

Palabras reservadas adicionales de C++

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

I.2.3. Tipos de errores en la programación

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

Ocasionados por un fallo de sintaxis en el código fuente.

No se genera el programa ejecutable.

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

using namespace std;

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

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

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



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

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

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



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

▷ **Errores lógicos (Logic errors)**

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

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

I.2.4. Cuidando la presentación

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

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

I.2.4.1. Escritura de código fuente

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

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



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

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



I.2.4.2. Etiquetado de las Entradas/Salidas

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

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



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



Las entradas de datos también deben etiquetarse adecuadamente:

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



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



I.3. Datos y tipos de datos

I.3.1. Representación en memoria de datos e instrucciones

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

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

Nos centramos en los datos.

I.3.2. Datos y tipos de datos

I.3.2.1. Declaración de datos

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

Nombre de empleado: Juan Pérez

Número de habitantes : 75225813

π : 3.14159

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

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

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

nombre_empleado	numero_habitantes	Pi
Juan Pérez	75225	3.1415927

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

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

```
double dato = 4.5;
```

produce el mismo resultado que:

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

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

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

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

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

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

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

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

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

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



v, l1, l2, hp



voltaje, lado1, lado2, hipotenusa

- ▷ No utilizaremos nombres genéricos



aux, vector1



copia, calificaciones

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



precioventapublico, tasaanual



precio_venta_publico, tasa_anual

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

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



pvp, tae

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



cuenta, cuentas, Cuenta



cuenta, coleccion_cuentas, cuenta_ppal

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

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



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

Nosotros seguiremos, en parte, las directrices indicadas por Google en *[Google C++ Style Guide](https://google.github.io/styleguide/cppguide.html)*

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

I.3.2.2. Inicialización de los datos

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

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

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

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

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

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

salario_bruto	retencion
?	?

salario_bruto	retencion
32538.0	4229.94

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

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



Imprimirá un valor indeterminado.

GLGO: Garbage input, garbage output



I.3.2.3. Literales

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

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

I.3.2.4. Datos constantes

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

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

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

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

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

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

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

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

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

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

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

Compare el anterior código con el siguiente:

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



Ventajas al usar constantes en vez de literales:

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

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

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

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

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

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

IMPORTANT

No debemos llevar al extremo la anterior recomendación:

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



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

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

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

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

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

    retencion = salario_bruto * FRACCION_RETENCION;

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

salario_bruto	FRACCION_RETENCION	retencion
32538.0	0.18	4229.94

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

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

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

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

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

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

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

Nota:

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

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

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

Una sintaxis de programa algo más completa:

```
[ /* Breve descripción en lenguaje natural
    de lo que hace el programa */ ]

[ Inclusión de recursos externos ]
[ using namespace std; ]

int main(){
    [Declaración de constantes]
    [Declaración de variables]

    [Sentencias del programa separadas por ;]
}
```

I.3.2.5. Ámbito de un dato

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

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

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

I.4. Operadores y expresiones

I.4.1. Expresiones

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

```
3  
lado1
```

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

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

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

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

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

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

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

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

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

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

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

$3 + 5$ es una expresión entera

$3.5 + 6.7$ es una expresión real

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

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

Imprime en pantalla:

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

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

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

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

I.4.2. Terminología en Matemáticas

Notaciones usadas con los operadores matemáticos:

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

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

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

$3 + 5$ x/y

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

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

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

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

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

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

I.4.3. Operadores en Programación

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

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

Tipos de operadores:

- ▷ Los definidos en el núcleo del compilador.

No hay que incluir ninguna biblioteca

Suelen usarse tokens simbólicos para su representación:

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

Los operadores binarios suelen ser infijos:

`3 + 5`

`lado * lado`

- ▷ Los definidos en bibliotecas externas.

Por ejemplo, `cmath`

Suelen usarse tokens alfanuméricos para su representación:

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

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

`sqrt(4.2)`

`sin(6.4)`

`pow(3 , 6)`

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

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

```
double dato;
```

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

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

En una sentencia de asignación

```
variable = <expresión>
```

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

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

El comportamiento de un tipo de dato viene dado por:

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

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

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

I.5.1. Los tipos de datos enteros

I.5.1.1. Representación de los enteros

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

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

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

0000000000110101

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

¿Cuántos datos distintos podemos representar?

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

$$0000000000110101 \neq 0000000000110110$$

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

I.5.1.2. Rango de los enteros

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

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

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

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

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

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

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

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

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

I.5.1.3. Literales enteros

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

53 -406778 0

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

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

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

I.5.1.4. Operadores

Operadores binarios

Los más usuales son:

+ - * / %

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

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

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

Operaciones usuales:

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

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

Nota:

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

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

es equivalente a la siguiente:

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

Algunos ejemplos de sentencias equivalentes:

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

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

Operadores unarios de incremento y decremento

++ y -- Unarios de notación postfija.

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

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

Por ejemplo:

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

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

Operador unario de cambio de signo

- Unario de notación prefija.

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

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

I.5.1.5. Expresiones enteras

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

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

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

Reglas de precedencia:

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

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

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

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

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

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

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

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

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

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

I.5.2. Los tipos de datos reales

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

- ▷ Parte entera de 4.56 \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:

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

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

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

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

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

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

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

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

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

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

Por tanto:

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

IMPORTANT

Especial cuidado tendremos con operaciones del tipo

Repita varias veces

Ir sumándole a una `variable_real` varios valores reales;

ya que los errores de aproximación se irán acumulando.



Ampliación:



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

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

I.5.2.3. Rango y Precisión

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

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

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

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

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

```
double valor_real;
```

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

```
valor_real = 11.0;
```

```
// Almacena: 11.0
```



Correcto

```
valor_real = 1.1;
```

```
// Almacena: 1.1000000000000001
```



Problema de redondeo

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

```
valor_real = 1000000000000000000100.0;
```

```
// Almacena: 1000000000000000000000.0
```



Problema de precisión

```
valor_real = 0.100000000000000000009;
```

```
// Almacena: 0.10000000000000001
```



Problema de precisión

```
valor_real = 1.000000000000000000009;
```

```
// Almacena: 1.0
```



Problema de precisión

```
valor_real = 10000000001.000000000004;
```

```
// Almacena: 10000000001.0
```



Problema de precisión

En resumen:

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

I.5.2.4. Indeterminación e Infinito

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

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

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

```
double valor_real, divisor = 0.0;

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

I.5.2.5. Operadores

Los operadores matemáticos usuales también se aplican sobre datos reales:

+, -, *, /

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

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

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

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

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

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

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

I.5.2.6. Funciones estándar

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

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

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

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

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

using namespace std;

int main(){
    double real, otro_real;

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

Nota:

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

I.5.2.7. Expresiones reales

Son expresiones cuyo resultado es un número real.

`sqrt(real)` es una expresión real

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

Precedencia de operadores en las expresiones reales:

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

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



Ejemplo. Construya una expresión para calcular la siguiente fórmula:

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

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

Ejercicio. Construya una expresión para calcular la distancia euclídea entre dos puntos del plano $P1 = (x_1, y_1)$, $P2 = (x_2, y_2)$. Puede usar la función `sqrt` pero no la función `pow`.

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

I.5.3. Operando con tipos numéricos distintos

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

I.5.3.1. Asignaciones entre datos de distinto tipo

Vamos a analizar la siguiente situación:

```
dato_del_tipo_A = dato_del_tipo_B;
```

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

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

Esta transformación es temporal (mientras se evalúa la expresión)

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

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

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

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

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



```
grande = 3600000000000000;
```

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



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



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

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

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

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

En resumen:

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

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

Ampliación:



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

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

using namespace std;

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

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

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

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

Vamos a analizar la siguiente situación:

```
variable_del_tipo_A = expresión_con_datos_del_tipo_A;
```

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

► Casos no problemáticos

Ejemplo. Suma de dos enteros pequeños

```
int chico;
```

```
chico = 2;
```

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



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

Ejemplo. Calcular la longitud de una circunferencia

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

```
double radio;
```

```
.....
```

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

```
// Sin problemas.
```



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

Ampliación:



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

`'B' - 'A'`

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

► Casos problemáticos

Ejemplo. Expresión fuera de rango

```
int chico;
```

```
chico = 2147483647;    // máximo int (32 bits). Sin problemas.
```

```
chico = chico + 1;    // máximo int (32 bits) + 1
```

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



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

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

Ejemplo. ¿Qué pasaría en este código?

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

grande = chico * chico;

// grande = 304084036    Error lógico 😞
```

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

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

Posibles soluciones: Lo veremos en la página **75**

Nota:

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

```
double real, otro_real;

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

I.5.3.3. Expresiones con datos numéricos de distinto tipo

Vamos a analizar la siguiente situación:

```
variable_de_cualquier_tipo = expresión_con_datos_de_tipo_cualquiera;
```

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

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

Esta transformación es temporal (mientras se evalúa la expresión)

► *Casos no problemáticos*

Ejemplo. Calcular la longitud de una circunferencia

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

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

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

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

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

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

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



► Casos problemáticos

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

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

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



Analicemos la situación:

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

Posibles soluciones (de peor a mejor)

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

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

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

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

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

IMPORTANT

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

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

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

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

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

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

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

En resumen:

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

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

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

I.5.3.4. El operador de casting (Ampliación)

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

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

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

Ejemplo. Media aritmética:

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

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



Ejemplo. Retomamos el ejemplo de la página 71:

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

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


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



¿Qué ocurre en el siguiente ejemplo?

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

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

// grande = 304084036 
```

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

Nota:

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

(<tipo de dato>) expresión

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

I.5.4. El tipo de dato cadena de caracteres

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

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

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

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

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

Escribiría en pantalla:

Bienvenidos

Empiezo a escribir en la siguiente línea

Acabo de tabular esta línea

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

Nota:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Nota:

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

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

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

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

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

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

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

Ampliación:

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

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



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

```
string cadena;
```

```
cadena = "";
```

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

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

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

I.5.5. El tipo de dato carácter

I.5.5.1. Representación de caracteres en el ordenador

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

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

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

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

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

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

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

modificación de la anterior.

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

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

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

Nota:

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

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

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

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

I.5.5.2. Literales de carácter

Son tokens formados por:

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

'!' 'A' 'a' 'ñ' '5'

Observe lo siguiente:

'5' es un literal de carácter

5 es un literal entero

"5" es un literal de cadena de caracteres

'cinco' no es un literal de carácter correcto

'11' no es un literal de carácter correcto

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

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

Ejemplo.

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

Salida en pantalla:

```
caña  
!  
B1  
    "Bye"
```

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

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

I.5.5.3. Asignación de literales de carácter

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

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

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

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

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

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

Ya sabemos que lo siguiente no es correcto:

```
int letra_piso;

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

int letra_piso;

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

I.5.5.4. El tipo de dato `char`

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

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

```
int letra_piso;

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

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

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

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

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

```
char letra_piso;

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

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

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

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

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

Ejemplo.

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

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

Escribiría en pantalla:

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

Esto es una comilla simple: '
```

Ampliación:

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

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

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



Ampliación:



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

Así pues, tenemos doble lío:

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

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

Ampliación:



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

I.5.5.5. Funciones estándar y operadores

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

```
                                tolower    toupper

#include <cctype>
using namespace std;

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

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

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

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

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

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

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

I.5.6. Lectura de varios datos

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

```
int entero, otro_entero;
double real;

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

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

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

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

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

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

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

Supongamos que el usuario introduce 43 52.1<Enter>

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

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

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

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

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

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

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

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

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

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

Contenido de fichero.txt:

```
43      52.1\n37
```

Desde un editor de texto se vería lo siguiente:

```
43      52.1
37
```

La lectura sería así:

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

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

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

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

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

□ □ a □ 1 2 3	<code>cin >> entero;</code> Fallo de lectura <code>cin >> lo_que_sea;</code> Fallo de lectura	□ □ 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 y saltos de línea que previamente hubiese:

```
char character;
```

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

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

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

Lo mismo ocurre si hubiese un carácter de nueva línea:

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

Ampliación:

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



I.5.7. El tipo de dato lógico o booleano

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

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

I.5.7.1. Rango

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

`true` `false`

I.5.7.2. Funciones estándar y operadores lógicos

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

Algunas funciones que devuelven un valor lógico:

▷ **En la biblioteca cctype:**

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

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

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

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

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

▷ **En la biblioteca cmath:**

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

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

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

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

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


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

$p \equiv$ Carlos es varón

$q \equiv$ Carlos es joven

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

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

Recordad la siguiente regla nemotécnica:

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

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

Tabla de Precedencia:

!

`&&`

`||`

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

I.5.7.3. Operadores Relacionales

Son los operadores habituales de comparación de expresiones numéricas.

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

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

Algunos ejemplos:

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

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

`!=` es el operador relacional distinto.

`!` es la negación lógica.

`==` es el operador relacional de igualdad.

`=` es la operación de asignación.

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

```
// Ejemplo de operadores relacionales

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

    entero1 = 3;
    entero2 = 5;

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

    real1 = 3.8;
    real2 = 8.1;

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

Veremos su uso en la *sentencia condicional*:

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

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

Tabla de Precedencia:

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

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

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

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

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



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

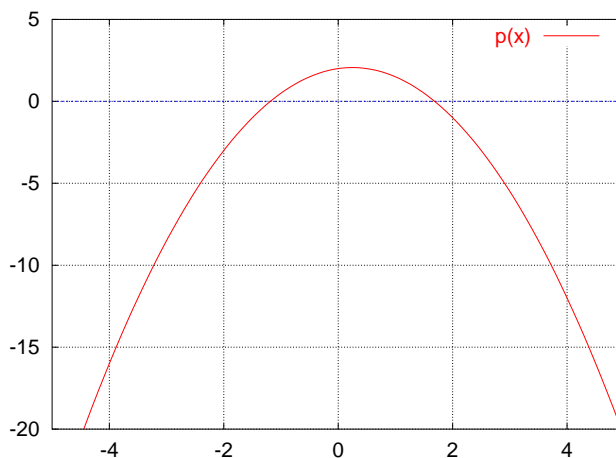


Ejercicio. Escriba una expresión lógica que devuelva `true` si un número entero `edad` está en el intervalo `[0,100]`

I.6. El principio de una única vez

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

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



Algoritmo: Raíces de una parábola

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

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

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

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

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

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

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



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

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

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

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

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

    // Cada expresión sólo se evalúa una vez:

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

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



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

Nota:

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

Principio de Programación:

Una única vez (Once and only once)

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



O dicho de una manera informal:

El código no debe estar repetido en distintos sitios del programa



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

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

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

Bibliografía recomendada para este tema:

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

Resúmenes:

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

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

En una sentencia de asignación

`variable = <expresión>`

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

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

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

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

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

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

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

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

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

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

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



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



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

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

IMPORTANT

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

IMPORTANT

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

IMPORTANT

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



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



El código no debe estar repetido en distintos sitios del programa



Principio de Programación:

Una única vez (Once and only once)

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



Tema II

Estructuras de control

Objetivos:

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

II.1. Estructura condicional

II.1.1. Flujo de control

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

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

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

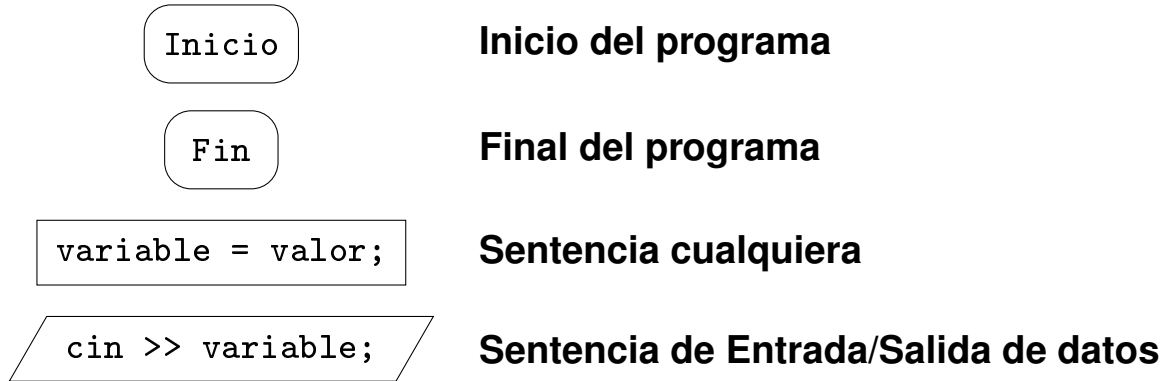
1  cout << "\nIntroduce coeficiente de 2º grado: ";
2  cin >> a;
3  cout << "\nIntroduce coeficiente de 1er grado: ";
4  cin >> b;
5  cout << "\nIntroduce coeficiente independiente: ";
6  cin >> c;

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

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

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

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



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

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

▷ **Hacia delante.**

Implicará que un conjunto de sentencias no se ejecutarán.

Vienen definidos a través de una estructura condicional.

▷ **Hacia atrás.**

Implicará que volverá a ejecutarse un conjunto de sentencias.

Vienen definidos a través de una estructura repetitiva.

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

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

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

II.1.2. Estructura condicional simple

II.1.2.1. Formato

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

<condición> es una expresión lógica

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

Los paréntesis que encierran la condición son obligatorios.

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

Ejemplo. Continuando el ejemplo de la página 119



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

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

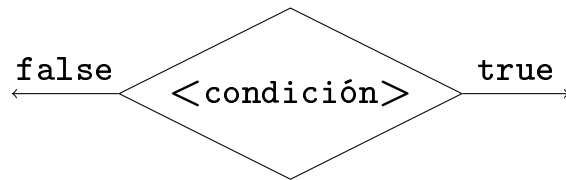
    cout << "\nIntroduce coeficiente de 2º grado: ";
    cin >> a;
    cout << "\nIntroduce coeficiente de 1er grado: ";
    cin >> b;
    cout << "\nIntroduce coeficiente independiente: ";
    cin >> c;

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

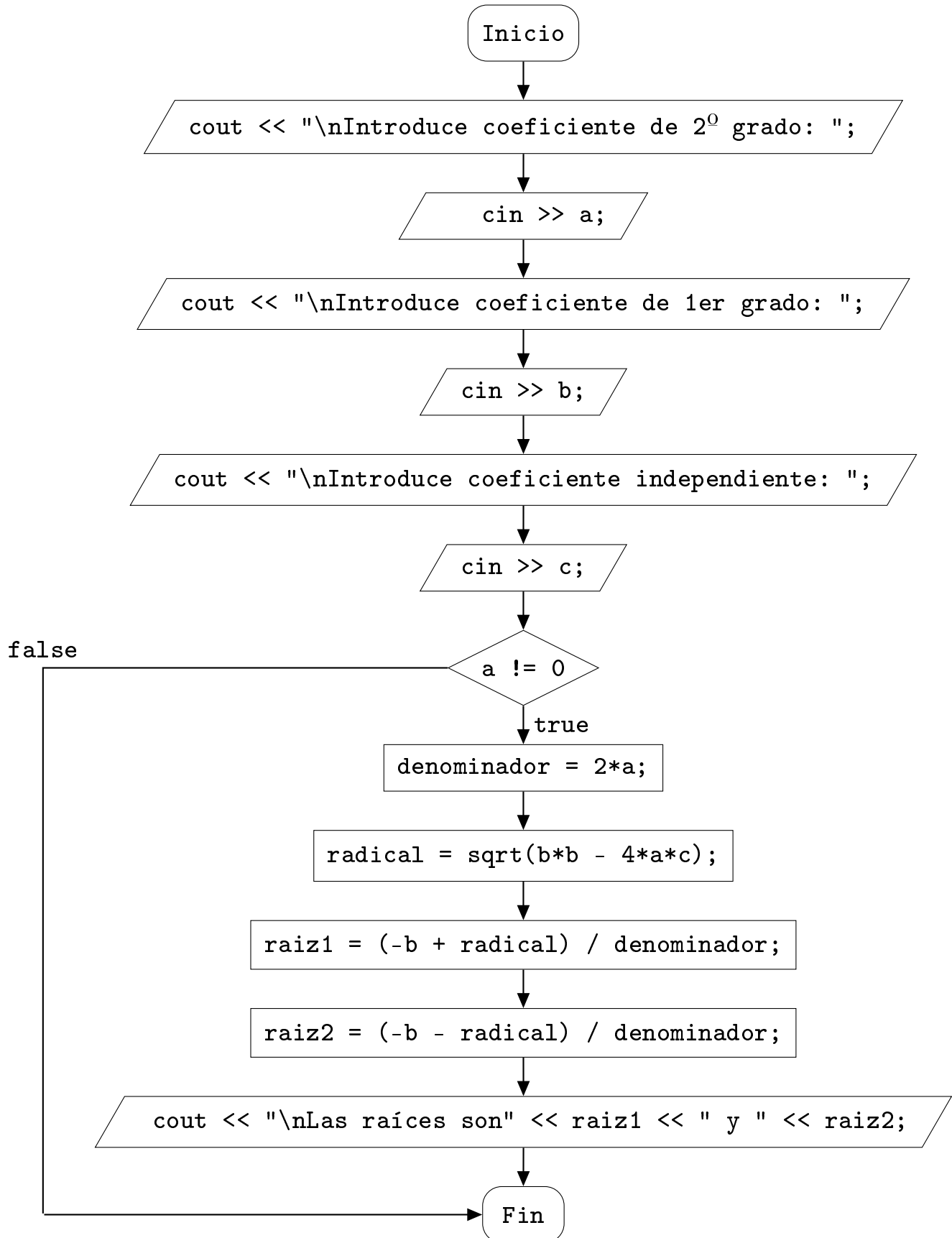
        cout << "\nLas raíces son" << raiz1 << " y " << raiz2;
    }
}
```

II.1.2.2. Diagrama de flujo

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



El ejemplo de la ecuación de segundo grado quedaría:



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

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

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

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

Para el compilador es como si fuese:

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

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

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

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

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

```
int entero;
cin >> entero;

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

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

o si se prefiere:

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

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

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

```
Fin del programa
```

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

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

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

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

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

```
Fin del programa
```


II.1.2.3. Variables no asignadas en los condicionales

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

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

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

```
cin >> a;
```

```
cin >> b;
```

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



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

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

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

```
cin >> a;
```

```
cin >> b;
```

```
max = b;
```

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



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

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



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

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

II.1.2.4. Cuestión de estilo

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

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

```
cin >> c;

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

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

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

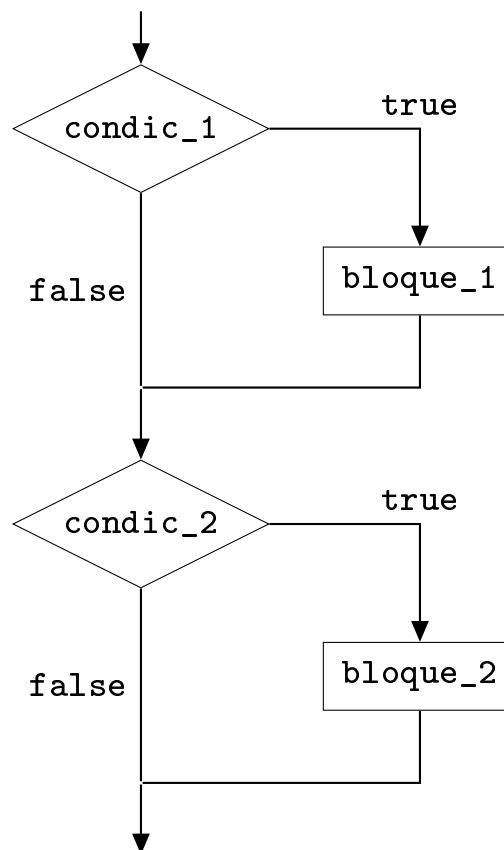
Destaque visualmente el bloque de instrucciones de una estructura condicional.

IMPORTANT

II.1.2.5. Estructuras condicionales consecutivas

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

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



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

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

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

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

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

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



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

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

```
salario_final = salario_base;

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

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

II.1.2.6. Condiciones compuestas

En muchos casos tendremos que utilizar condiciones compuestas:

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

```
int edad;
double salario_base, salario_final;

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

salario_final = salario_base;

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



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

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

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



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

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

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

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

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

```
cin >> inferior;
```

```
cin >> superior;
```

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

```
cin >> dato;
```

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

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



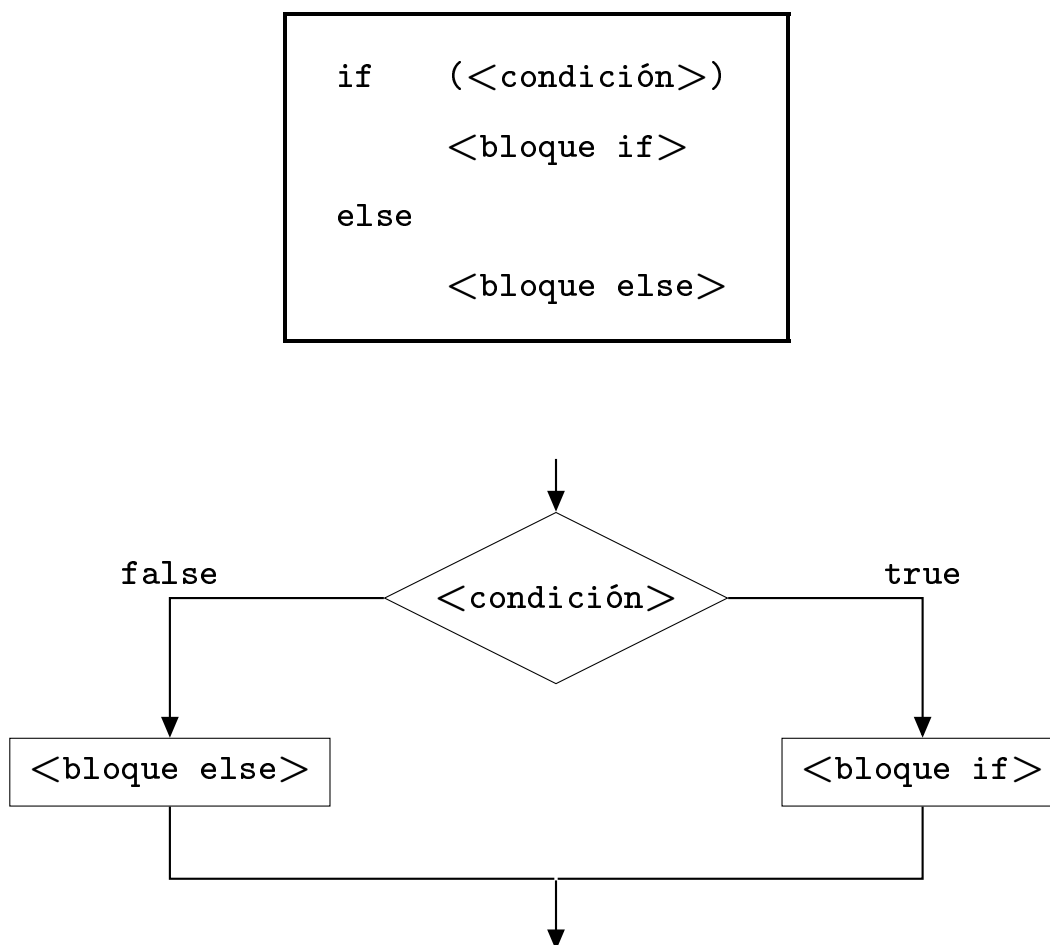
136

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

II.1.3. Estructura condicional doble

II.1.3.1. Formato

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



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

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



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

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

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

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

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

http://decsai.ugr.es/jccubero/FP/II_Ec2Grado_vs0.cpp

Ejemplo. Compruebe si un número es par (continuación)

```
cin >> entero;

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

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

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

Ejemplo. Compruebe si el salario es bajo (continuación)

```
double salario;

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

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

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

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

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

Ejemplo. Retomemos el ejemplo de la página 135:

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

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

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



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

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

```
pertenece_al_intervalo = false;

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

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

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

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

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



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



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

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

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

Observe que no es necesario poner

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

Basta con poner:

```
if (pertenece_al_intervalo)
```

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



II.1.3.2. Condiciones mutuamente excluyentes

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

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

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

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



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

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

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

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

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



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

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

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



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

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

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

Ejemplo.

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

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

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

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



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

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

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



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

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

```
edad >= 45 && salario_base < 1300 -> +4%  
Otro caso -> +1%
```

¿Cuándo se produce la subida del 1%? Cuando la expresión `edad >= 45 && salario_base < 1300` es false, es decir, cuando cualquiera de ellos es false. Nos quedaría:

```
edad >= 45 && salario_base < 1300 -> +4%  
edad < 45 || salario_base >= 1300 -> +1%
```

¿Sería correcto entonces poner lo siguiente?

```
salario_final = salario_base;  
  
if (edad >= 45 && salario_base < 1300)  
    salario_final = salario_final * 1.04;  
if (edad < 45 || salario_base >= 1300) // Código repetido  
    salario_final = salario_final * 1.01;
```



El resultado, de cara al usuario, sería el mismo, pero es un código nefasto, ya que se viola el principio de una única vez. Por lo tanto, usamos mucho mejor una estructura condicional doble:

```
salario_final = salario_base;  
  
if (edad >= 45 && salario_base < 1300)  
    salario_final = salario_final * 1.04;  
else  
    salario_final = salario_final * 1.01;
```



En resumen:

La estructura condicional doble nos permite trabajar con dos condiciones mutuamente excluyentes, comprobando únicamente una de ellas en la parte del `if`. Esto nos permite cumplir el principio de una única vez.

<code>if (cond)</code>		<code>if (cond)</code>
<code>...</code>		<code>...</code>
<code>if (!cond)</code>	→	<code>else</code>
<code>...</code>		<code>...</code>

II.1.3.3. Álgebra de Boole

Debemos intentar simplificar las expresiones lógicas, para que sean fáciles de entender. Usaremos las propiedades del *álgebra de Boole (Boolean algebra)* :

$\neg(A \vee B)$	equivale a	$\neg A \wedge \neg B$
$\neg(A \wedge B)$	equivale a	$\neg A \vee \neg B$
$A \wedge (B \vee C)$	equivale a	$(A \wedge B) \vee (A \wedge C)$
$A \vee (B \wedge C)$	equivale a	$(A \vee B) \wedge (A \vee C)$

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

Ampliación:

Consulte:

http://es.wikipedia.org/wiki/Algebra_booleana

<http://serbal.pntic.mec.es/~cmunoz11/boole.pdf>



Ejemplo. Es un contribuyente especial si la edad está entre 16 y 65 y sus ingresos están por debajo de 7000 o por encima de 75000 euros. La expresión lógica sería:

```
(16 <= edad && edad <= 65 && ingresos < 7000)
||
(16 <= edad && edad <= 65 && ingresos > 75000)
```

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

```
(16 <= edad && edad <= 65)
&&
(ingresos < 7000 || ingresos > 75000)
```

De esta forma, no repetimos la expresión `edad >= 16 && edad <= 65`

En resumen:

Utilice las leyes del Álgebra de Boole para simplificar las expresiones lógicas.

Ejemplo. Un trabajador es un aprendiz si su edad no está por debajo de 16 (estricto) o por encima de 18. La expresión lógica que debemos usar es:

```
!(edad < 16 || 18 <= edad)      // Difícil de entender
```

Aplicando las leyes de Morgan:

```
!(edad < 16 || 18 <= edad)  
    equivale a  
!(edad < 16) && !(18 <= edad)  
    equivale a  
(16 <= edad) && (edad < 18)
```

Realmente no son necesarios los paréntesis de las expresiones ya que el operador && tiene menos prioridad. Nos quedaría finalmente:

```
16 <= edad && edad < 18      // Más fácil de entender
```

Ejemplo. Suba un 4% el salario del trabajador si tiene más de 45 años. Si no los tiene, también se subirá siempre y cuando tenga más de 3 años de experiencia.

```
salario_final = salario_base;

if (edad >= 45 || (edad < 45 && experiencia > 3))
    salario_final = salario_final * 1.04;
```

Observe que `edad >= 45` **equivale a** `!(edad < 45)` **por lo que estamos ante una expresión del tipo**

`A || (!A && B)`

Veamos que esta expresión se puede simplificar a la siguiente:

`A || B`

Aplicamos la ley distributiva:

`A || (!A && B) <-> (A || !A) && (A || B)`
`<->`
`True && (A || B) <-> A || B`

Aplicándolo a nuestro ejemplo, nos quedaría finalmente:

```
salario_final = salario_base;

if (edad >= 45 || experiencia > 3)
    salario_final = salario_final * 1.04;
```

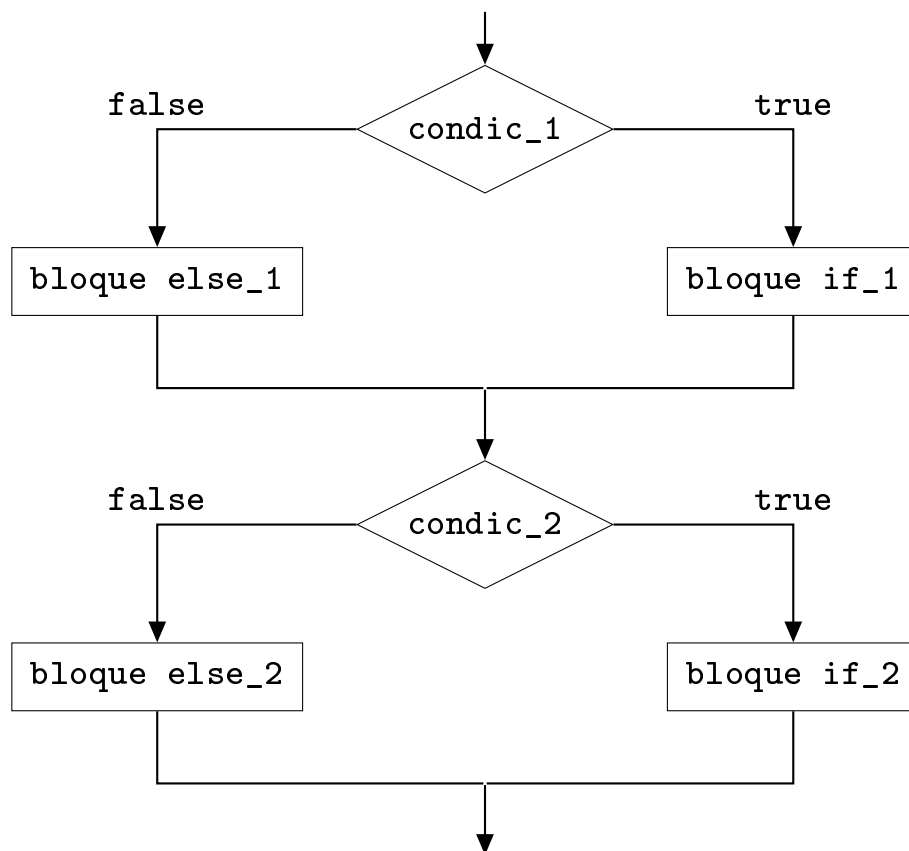
La expresión `A || (!A && B)` ***la sustituiremos por la equivalente a ella:*** `A || B`

II.1.3.4. Estructuras condicionales dobles consecutivas

Un condicional doble garantiza la ejecución de un bloque de instrucciones: o bien el bloque del `if`, o bien el bloque del `else`. Si hay dos condicionales dobles consecutivos, se ejecutarán los bloques de instrucciones que corresponda: o ninguno, o uno de ellos, o los dos.

```
if (condic_1)
    bloque if_1
else
    bloque else_1
```

```
if (condic_2)
    bloque if_2
else
    bloque else_2
```



Veamos un ejemplo de un condicional doble y otro simple consecutivos.

Ejemplo. Retome el ejemplo de la subida salarial de la página 146. Considere otro criterio en la subida salarial: si tiene más de dos hijos, se subirá un 2% adicional sobre el salario incrementado anteriormente.

```
edad >= 45 && salario_base < 1300  ->  +4%
Otro caso                          ->  +1%

numero_hijos > 2                   ->  +2%
```

Nos quedaría:

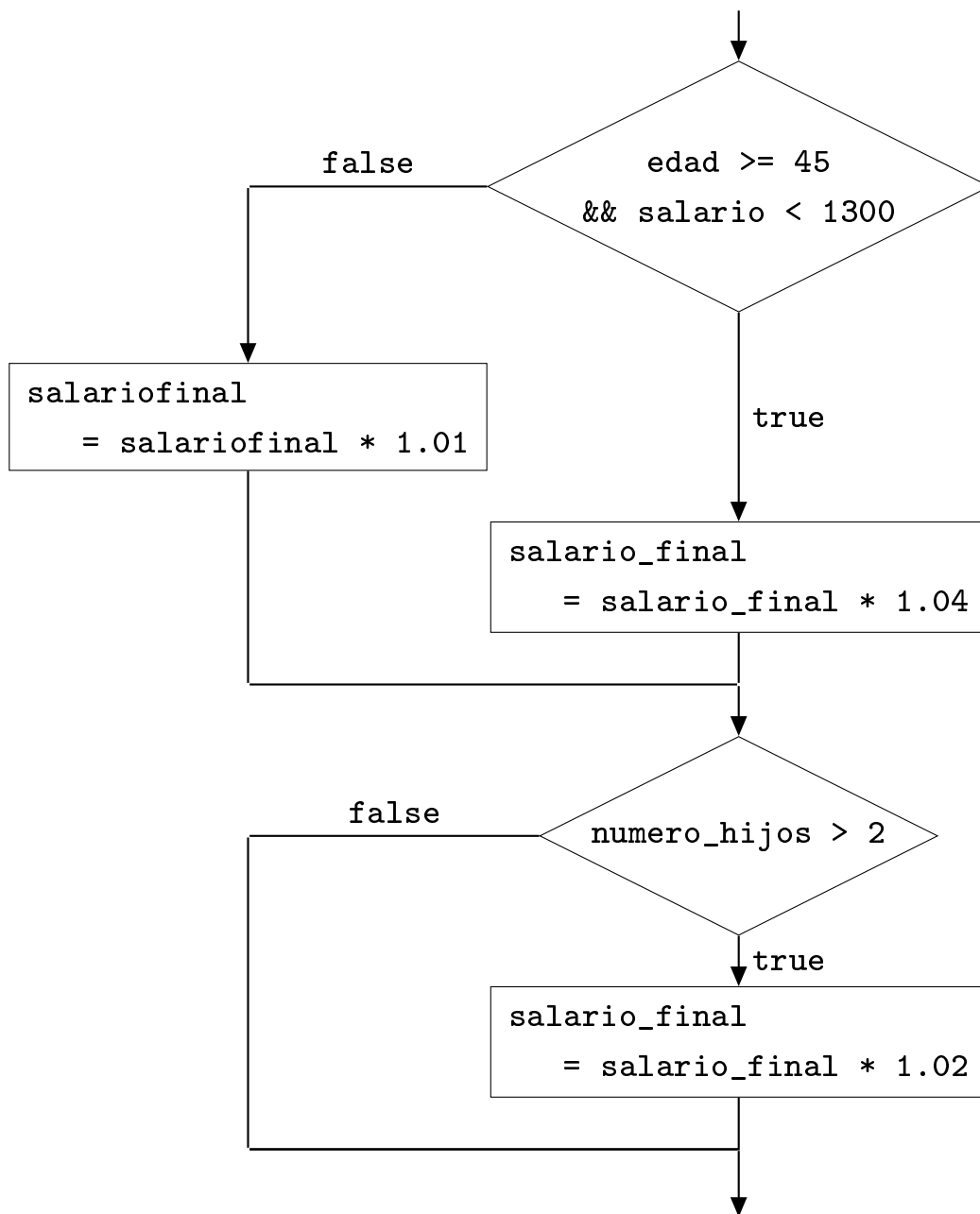
```
.....
salario_final = salario_base;

if (edad >= 45 && salario_base < 1300)
    salario_final = salario_final * 1.04;
else
    salario_final = salario_final * 1.01;

if (numero_hijos > 2)
    salario_final = salario_final * 1.02;
```



Destaquemos que la segunda subida se realiza sobre el salario final ya modificado en el primer condicional. Se deja como ejercicio realizar la segunda subida tomando como referencia el salario base.



Veamos un ejemplo de dos condicionales dobles consecutivos.

Ejemplo. Lea la edad y la altura de una persona y diga si es mayor o menor de edad y si es alta o no. Se pueden dar las cuatro combinaciones posibles: mayor de edad alto, mayor de edad no alto, menor de edad alto, menor de edad no alto.

```
int edad, altura;

cin >> edad;
cin >> altura;

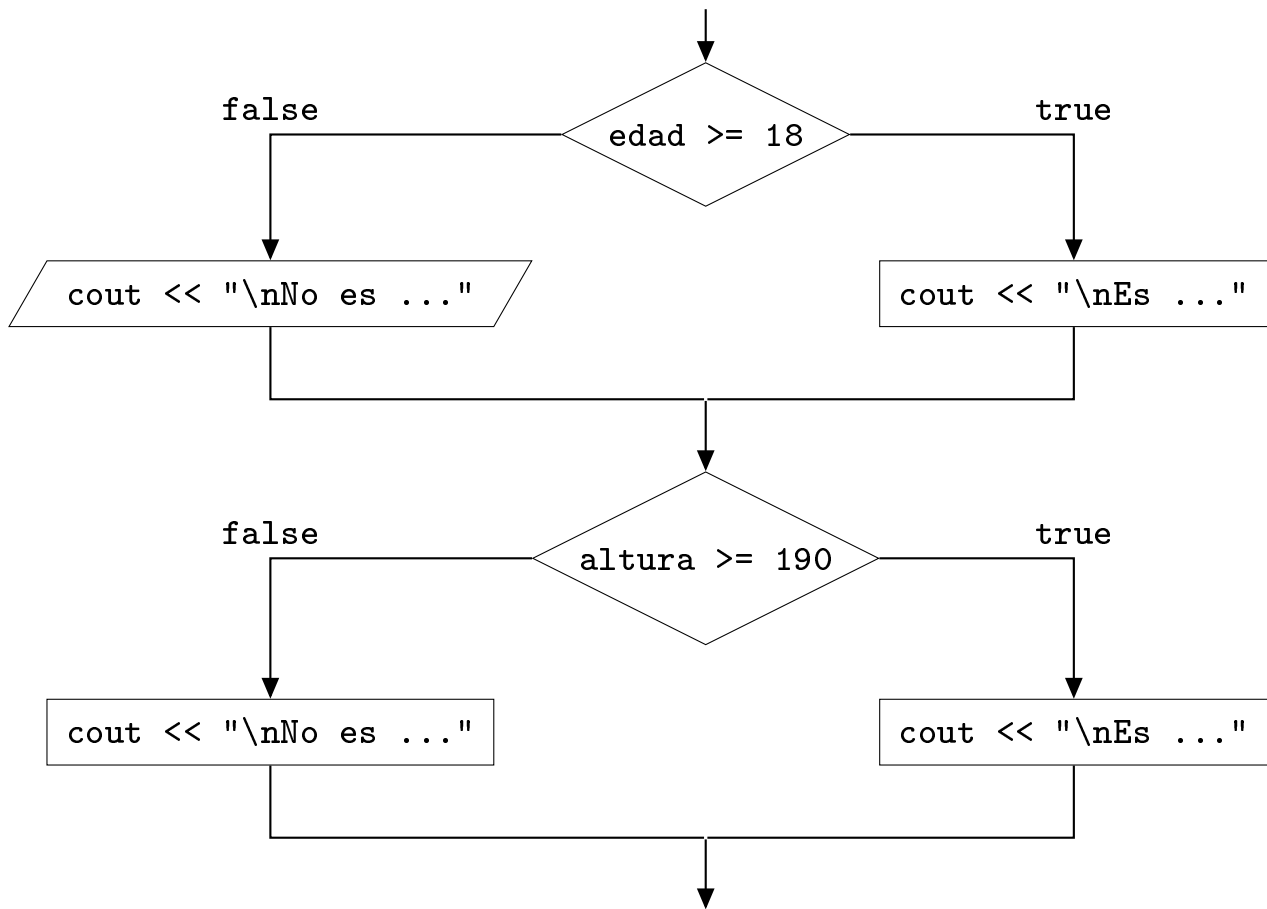
if (edad >= 18)
    cout << "\nEs mayor de edad";
else
    cout << "\nEs menor de edad";

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



Siempre se imprimirán dos mensajes. Uno relativo a la condición de ser mayor o menor de edad y el otro relativo a la altura.

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



Tal y como veíamos en la página 143, si hubiésemos duplicado las expresiones de los condicionales habría mucho código repetido y estaríamos violando el principio de una única vez:

```
if (edad >= 18 && altura >= 190)
    cout << "\nEs mayor de edad" << "\nEs alto/a";

if (edad >= 18 && altura < 190)
    cout << "\nEs mayor de edad" << "\nNo es alto/a";

if (edad < 18 && altura >= 190)
    cout << "\nEs menor de edad" << "\nEs alto/a";

if (edad < 18 && altura < 190)
    cout << "\nEs menor de edad" << "\nNo es alto/a";
```

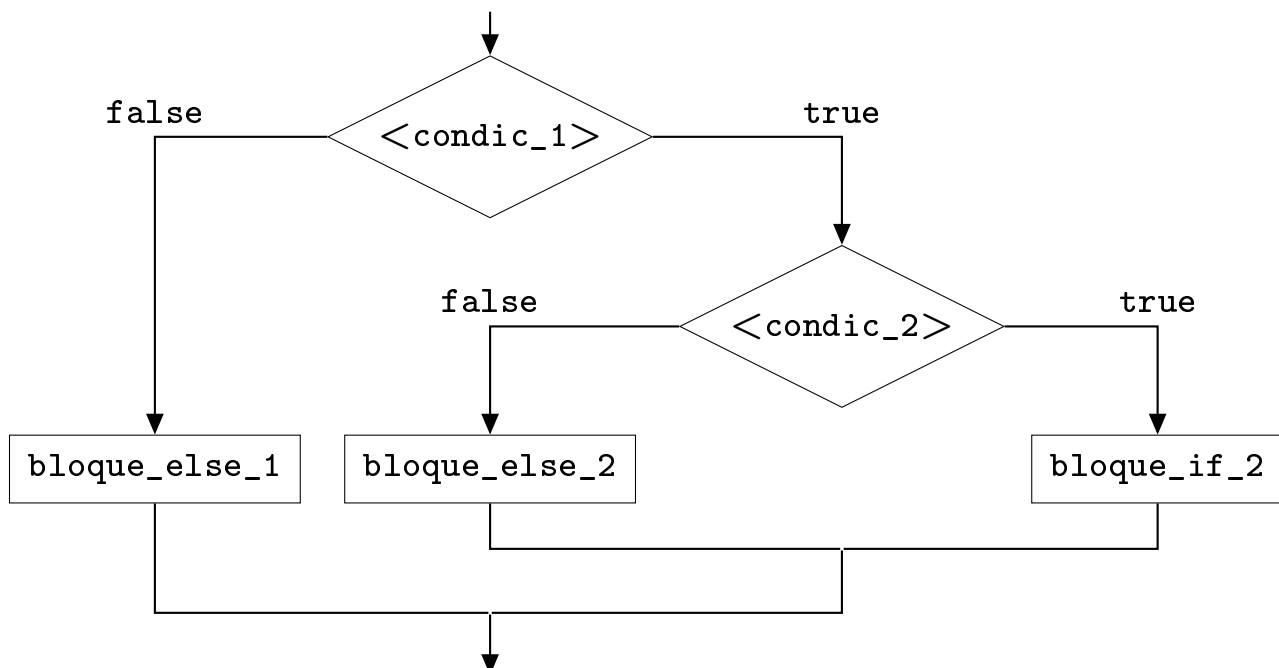


II.1.4. Anidamiento de estructuras condicionales

II.1.4.1. Funcionamiento del anidamiento

Dentro de un bloque `if` o `else`, puede incluirse otra estructura condicional, anidándose tanto como permita el compilador (aunque lo normal será que no tengamos más de tres o cuatro niveles de anidamiento). Por ejemplo, si anidamos un condicional doble dentro de un bloque `if`:

```
if (condic_1){  
    if (condic_2){  
        bloque_if_2  
    }  
    else{  
        bloque_else_2  
    }  
else{  
    bloque_else_1  
}
```



Ejemplo. ¿Cuándo se ejecuta cada instrucción?

```
if (condic_1){  
    inst_1;  
  
    if (condic_2){  
        inst_2;  
    }  
    else{  
        inst_3;  
    }  
  
    inst_4;  
}  
else{  
    inst_5;  
  
    if (condic_3)  
        inst_6;  
}
```

	condic_1	condic_2	condic_3
inst_1	true	da igual	da igual
inst_2	true	true	da igual
inst_3	true	false	da igual
inst_4	true	da igual	da igual
inst_5	false	da igual	da igual
inst_6	false	da igual	true

Recordemos que las estructuras condicionales consecutivas se utilizan cuando los criterios de las condiciones son independientes. Por contra, Usaremos los condicionales anidados cuando los criterios sean dependientes entre sí.

Ejemplo. Cambiamos el ejemplo de la página 155 para implementar el siguiente criterio: Si es mayor de edad, el umbral para decidir si una persona es alta o no es 190 cm. Si es menor de edad, el umbral baja a 175 cm.

Ahora, el criterio por el que determinamos si una persona es alta depende de la edad. Por tanto, en vez de usar dos condicionales consecutivos como hicimos en el ejemplo de la página 155, ahora debemos usar condicionales anidados:

```
int edad, altura;

cin >> edad;
cin >> altura;

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

    if (altura >= 190)
        cout << "\nEs alto/a";
    else
        cout << "\nNo es alto/a";
}
else{                                     // <- edad < 18
    cout << "\nEs menor de edad";

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



Ejemplo. Retomemos el ejemplo de la subida salarial de la página 153. La subida por número de hijos se aplicaba a todo el mundo:

```
edad >= 45 && salario_base < 1300  ->  +4%
Otro caso                           ->  +1%

numero_hijos > 2                     ->  +2%
```

Ahora cambiamos el criterio para que la subida por número de hijos **sólo** se aplique en el caso de que se haya aplicado la primera subida (por la edad y salario)

```
edad >= 45 && salario_base < 1300  ->  +4%
    numero_hijos > 2                ->  +2%
Otro caso                           ->  +1%
```

Nos quedaría:

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

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

    if (numero_hijos > 2)
        salario_final = salario_final * 1.02;
}
else
    salario_final = salario_final * 1.01;
```



198

Ejemplo. Refinamos el programa para el cálculo de las raíces de una ecuación de segundo grado visto en la página 138 para contemplar los casos especiales.

```
.....
if (a != 0) {
    denominador = 2*a;
    radicando = b*b - 4*a*c;

    if (radicando == 0){
        raiz1 = -b / denominador;
        cout << "\nSólo hay una raíz doble: " << raiz1;
    }
    else{
        if (radicando > 0){
            radical = sqrt(radicando);
            raiz1 = (-b + radical) / denominador;
            raiz2 = (-b - radical) / denominador;
            cout << "\nLas raíces son" << raiz1 << " y " << raiz2;
        }
        else
            cout << "\nNo hay raíces reales.";
    }
}
else{
    if (b != 0){
        raiz1 = -c / b;
        cout << "\nEs una recta. La única raíz es " << raiz1;
    }
    else
        cout << "\nNo es una ecuación.";
}
```



Ejercicio. Lea sobre una variable `nota_escrito` la nota de que haya sacado un alumno de FP en el examen escrito. Súbale 0.5 puntos, siempre que haya sacado más de un 4.5. Debe controlar que la nota final no sea mayor de 10, de forma que todos los que hayan sacado entre un 9.5 y un 10, su nota final será de 10.

II.1.4.2. Anidar o no anidar: he ahí el dilema

Nos preguntamos si, en general, es mejor anidar o no. La respuesta es que depende de la situación.

Un ejemplo en el que es mejor no anidar

Ejemplo. Retomemos el ejemplo de la subida salarial de la página 161. Simplificamos el criterio: la subida de sueldo se hará cuando el trabajador tenga una edad mayor o igual de 45 y un salario por debajo de 1300. En otro caso se sube un 1%

```
edad >= 45 && salario_base < 1300  ->  +4%
En otro caso                        ->  +1%
```

.....

```
salario_final = salario_base;
```

```
if (edad >= 45 && salario_base < 1300)
    salario_final = salario_final * 1.04;
else
    salario_final = salario_final * 1.01;
```



Pero observe que también podríamos haber puesto lo siguiente:

```
if (edad >= 45)
    if (salario_base < 1300)
        salario_final = salario_final * 1.04;
    else
        salario_final = salario_final * 1.01;
else
    salario_final = salario_final * 1.01;
```



En general, las siguientes estructuras son equivalentes:

<pre>if (c1 && c2) bloque_A else bloque_B</pre>	<pre>if (c1) if (c2) bloque_A else bloque_B else bloque_B</pre>
---	---

Para demostrarlo, basta ver bajo qué condiciones se ejecutan los distintos bloques y comprobamos que son las mismas:

Primer caso: bloque_A: c1 true
 c2 true
 bloque_B: (c1&& c2) false:
 c1 true y c2 false
 c1 false y c2 true
 c1 false y c2 false

Segundo caso: bloque_A: c1 true
 c2 true
 bloque_B: c1 true y c2 false
 o bien c1 false

Son equivalentes. Elegimos la primera opción porque la segunda repite código (bloque_B)

Un ejemplo en el que es mejor anidar

Ejemplo. Retomemos el ejemplo de la subida salarial de la página 161. Cambiamos el criterio de subida por el siguiente: todo sigue igual salvo que las subidas de sueldo se harán sólo cuando el trabajador tenga una experiencia de más de dos años. En caso contrario, se le baja el salario un 1%

```
experiencia > 2
    edad >= 45 && salario_base < 1300  ->  +4%
        numero_hijos > 2                ->  +2%
    En otro caso                        ->  +1%
En otro caso                          ->  -1%
```

.....

```
salario_final = salario_base;
```

```
if (experiencia > 2){
    if (edad >= 45 && salario_base < 1300){
        salario_final = salario_final * 1.04;

        if (numero_hijos > 2)
            salario_final = salario_final * 1.02;
    }
    else
        salario_final = salario_final * 1.01;
}
else
    salario_final = salario_final * 0.99;
```



198

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

Si construimos las negaciones correspondientes al `else` de cada condición, el código anterior es equivalente al siguiente:

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

if (experiencia > 2 && edad >= 45 && salario_base < 1300)
    salario_final = salario_final * 1.04;
if (experiencia > 2 && edad >= 45
    && salario_base < 1300 && numero_hijos > 2)
    salario_final = salario_final * 1.02;
if (experiencia > 2 && (edad < 45 || salario_base >= 1300))
    salario_final = salario_final * 1.01;
if (experiencia <= 2)
    salario_final = salario_final * 0.99;
```



Obviamente, el anterior código es nefasto ya que no cumple el principio de una única vez.

<pre>if (c1 && c2 && c3) <accion_A> if (c1 && c2 && !c3) <accion_B> if (c1 && !c2) <accion_C> if (!c1) <accion_D></pre>	→	<pre>if (c1) if (c2) if (c3) <accion_A> else <accion_B> else <accion_C> else <accion_D></pre>
---	---	---



Podemos resumir lo visto en el siguiente consejo (que es una generalización del visto en la página 147)

Usaremos los condicionales dobles y anidados de forma coherente para no duplicar ni el código de las sentencias ni el de las expresiones lógicas de los condicionales, cumpliendo así el principio de una única vez.

Otro ejemplo en el que es mejor anidar

Ejemplo. Leemos dos enteros y presentamos un menú de operaciones al usuario.

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

int main(){
    double dato1, dato2, resultado;
    char opcion;

    cout << "\nIntroduce el primer operando: ";
    cin >> dato1;
    cout << "\nIntroduce el segundo operando: ";
    cin >> dato2;
    cout << "\nElija (S)Sumar, (R)Restar, (M)Multiplicar: ";
    cin >> opcion;
    opcion = toupper(opcion);

    if (opcion == 'S')
        resultado = dato1 + dato2;
    if (opcion == 'R')
        resultado = dato1 - dato2;
    if (opcion == 'M')
        resultado = dato1 * dato2;
    if (opcion != 'R' && opcion != 'S' && opcion != 'M')
        resultado = NAN;    // <- Hay que incluir cmath

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



Las condiciones `opcion == 'S'`, `opcion == 'R'`, etc, son mutuamente excluyentes entre sí. Cuando una sea `true` las otras serán `false`, pero estamos obligando al compilador a evaluar innecesariamente dichas condiciones.

Para resolverlo usamos estructuras condicionales dobles anidadas:

```
if (opcion == 'S')
    resultado = dato1 + dato2;
else
    if (opcion == 'R')
        resultado = dato1 - dato2;
    else
        if (opcion == 'M')
            resultado = dato1 * dato2;
        else
            resultado = NAN;
```

Una forma también válida de tabular es la siguiente:

```
if (opcion == 'S')
    resultado = dato1 + dato2;
else if (opcion == 'R')
    resultado = dato1 - dato2;
else if (opcion == 'M')
    resultado = dato1 * dato2;
else
    resultado = NAN;    // <- Hay que incluir cmath
```



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

También podríamos haber asociado las opciones a los caracteres
'+', '-', '*'

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

Cuando c1, c2 y c3 sean mutuamente excluyentes:

if (c1)		if (c1)
...		...
if (c2)		else if (c2)
...	→	...
if (c3)		else if (c3)
...		...



II.1.5. Estructura condicional múltiple

Retomemos el ejemplo de lectura de dos enteros y una opción de la página 170. Cuando tenemos que comprobar condiciones mutuamente excluyentes sobre un dato entero, podemos usar otra estructura alternativa.

```
switch (<expresión>) {  
    case <constante1>:  
        <sentencias1>  
        break;  
    case <constante2>:  
        <sentencias2>  
        break;  
    .....  
    [default:  
        <sentencias>]  
}
```

- ▷ <expresión> es un expresión entera.
- ▷ <constante i> es un literal entero.
- ▷ switch sólo comprueba la igualdad.
- ▷ No debe haber dos cases con la misma <constante> en el mismo switch. Si esto ocurre, sólo se ejecutan las sentencias del caso que aparezca primero.
- ▷ El identificador especial default permite incluir un caso por defecto, que se ejecutará si no se cumple ningún otro. Lo pondremos al final de la estructura como el último de los casos.

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

es equivalente a:

```
switch (opcion){
    case 'S':
        resultado = dato1 + dato2;
        break;
    case 'R':
        resultado = dato1 - dato2;
        break;
    case 'M':
        resultado = dato1 * dato2;
        break;
    default:
        resultado = NAN;
}
```

El gran problema con la estructura `switch` es que el programador olvidará en más de una ocasión, incluir la sentencia `break`. La única *ventaja* es que se pueden realizar las mismas operaciones para varias constantes:

```
cin >> opcion;

switch (opcion){
    case 'S':
    case 's':
        resultado = dato1 + dato2;
        break;
    case 'R':
    case 'r':
        resultado = dato1 - dato2;
        break;
    case 'M':
    case 'm':
        resultado = dato1 * dato2;
        break;
    default:
        resultado = NAN;
}
```

El compilador no comprueba que cada case termina en un break. Es por ello, que debemos evitar, en la medida de lo posible, la sentencia switch y usar condicionales anidados en su lugar.

II.1.6. Ámbito de un dato (revisión)

En la página 39 se introdujo el concepto de ámbito de un dato como los sitios en los que se conoce un dato.

Hasta ahora, los datos los hemos declarado al inicio del programa. Vamos a ver que esto no es una restricción del lenguaje. De hecho, podemos declarar datos prácticamente en cualquier lugar: por ejemplo, dentro de un bloque condicional. Su ámbito será únicamente dicho bloque, es decir, el dato no se conocerá fuera del bloque.

Ejemplo. Retomemos el ejemplo de la página 138. Podemos declarar las variables `radical` y `denominador` dentro del condicional, siempre que no se necesiten fuera de él.

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

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

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

    if (a != 0) {
        double radical, denominador;

        denominador = 2*a;
```



210

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

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

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

// cout << denominador;  <- Error de compilación
}
```

Por ahora, no abusaremos de la declaración de datos dentro de un bloque condicional. Lo haremos sólo cuando sea muy evidente que los datos declarados no se necesitarán fuera del bloque.

II.1.7. Algunas cuestiones sobre condicionales

II.1.7.1. Cuidado con la comparación entre reales

La expresión $1.0 == (1.0/3.0)*3.0$ podría evaluarse a false debido a la precisión finita para calcular $(1.0/3.0)$ (0.33333333)

Otro ejemplo:

```
double raiz_de_dos;

raiz_de_dos = sqrt(2.0);

if (raiz_de_dos * raiz_de_dos != 2.0)    // Podría evaluarse a true
    cout << ";Raíz de dos al cuadrado no es igual a dos!";
```

Otro ejemplo:

```
double descuento_base, porcentaje;

descuento_base = 0.1;
porcentaje = descuento_base * 100;

if (porcentaje == 10.0)    // Podría evaluarse a false :-0
    cout << "Descuento del 10%";
```

Recuerde que la representación en coma flotante no es precisa, por lo que 0.1 será internamente un valor próximo a 0.1, pero no igual.

Soluciones:

- ▷ **Fomentar condiciones de desigualdad cuando sea posible.**
- ▷ **Fijar un *error* de precisión y aceptar igualdad cuando la diferencia de las cantidades sea menor que dicho error.**

```
double real1, real2;  
const double epsilon = 0.00001;
```

```
if (real1 - real2 < epsilon)  
    cout << "Son iguales";
```

Para una solución aún mejor consulte:

[https://randomascii.wordpress.com/2012/02/25/
comparing-floating-point-numbers-2012-edition/](https://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/)

II.1.7.2. Evaluación en ciclo corto y en ciclo largo

En una condición compuesta del tipo `A && B && C`, si la expresión `A` fuese `false`, ya no sería necesario evaluar ni `B`, ni `C`. Lo mismo ocurriría si la expresión fuese del tipo `A || B || C` y `A` fuese `true`. Los compiladores pueden analizar a priori una expresión compuesta y dejar de evaluar las expresiones que la constituyen en cuanto ya no sea necesario.

Evaluación en ciclo corto (Short-circuit evaluation) : El compilador optimiza la evaluación de expresiones lógicas evaluando sus términos de izquierda a derecha hasta que ya no sea necesario. La mayoría de los compiladores realizan este tipo de evaluación por defecto.

Evaluación en ciclo largo (Eager evaluation) : El compilador evalúa todos los términos de la expresión lógica para conocer el resultado de la expresión completa.

Ejemplo.

```
if (n != 0 && total/n < tope) ...
```

Supongamos que `n` es cero.

- ▷ **Ciclo largo**: El compilador evalúa (innecesariamente) todas las expresiones lógicas. En este caso, se podría producir un error al dividir por cero.
- ▷ **Ciclo corto**: El compilador evalúa sólo la primera expresión lógica.

Ejercicio. Re-escriba el anterior ejemplo (evitando dividir por cero) usando otra estructura que no involucre una condición compuesta. Para resolverlo, observe que la condición `total/n < tope` es dependiente de la condición `n != 0`, en el sentido de que si la segunda es `false`, no queremos evaluar la otra.

II.1.8. Programando como profesionales

II.1.8.1. Diseño de algoritmos fácilmente extensibles

Ejemplo. Calcule el mayor de tres números a, b, c

La primera idea podría ser contemplar todas las posibilidades de orden:

$$b \leq a \leq c, a \leq b \leq c, a \leq c \leq b, \dots$$

y construir un condicional `if` para cada una de ellas. Esto generaría demasiadas expresiones lógicas:

Algoritmo: Mayor de tres números. Versión 1

▷ **Idea:** Analizando en qué situación(es) el máximo es a, b o c .

▷ **Entradas:** a, b y c

Salidas: El mayor entre a, b y c

▷ **Descripción:**

Si a es mayor que los otros, el mayor es a

Si b es mayor que los otros, el mayor es b

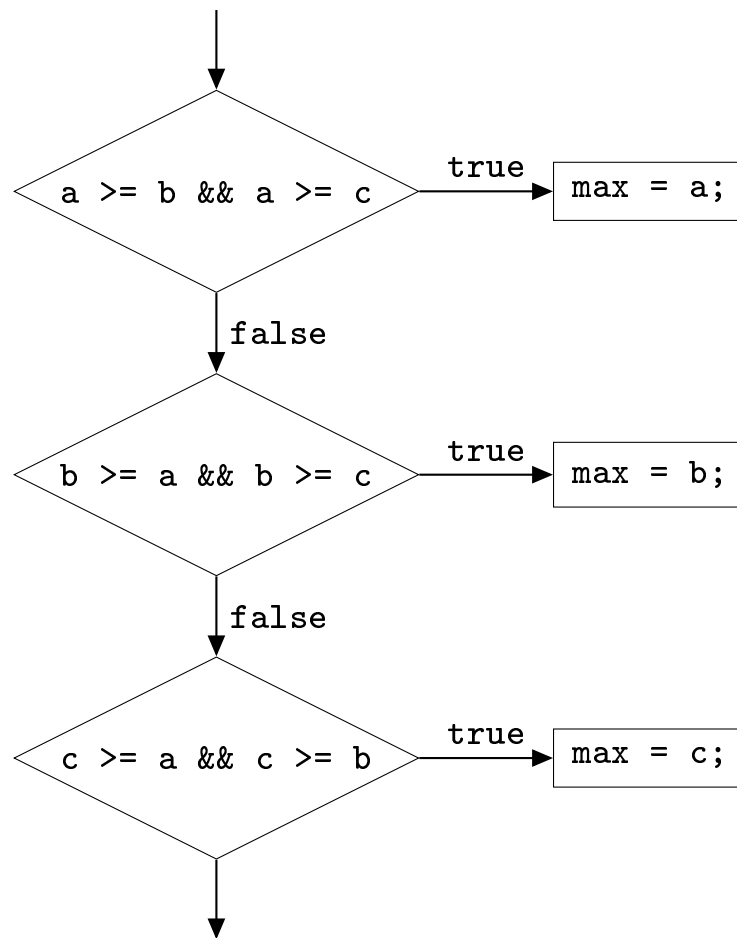
Si c es mayor que los otros, el mayor es c

▷ **Implementación:**

```
if ((a >= b) && (a >= c))
    max = a;
if ((b >= a) && (b >= c))
    max = b;
if ((c >= a) && (c >= b))
    max = c;
```



187



Inconvenientes:

- ▷ Al tener tres condicionales seguidos, siempre se evalúan las 6 expresiones lógicas. En cuanto hallemos el máximo deberíamos parar y no seguir preguntando.
- ▷ Podemos resolver el problema usando menos de 6 expresiones lógicas.

Ejemplo. El mayor de tres números (segunda aproximación)

Algoritmo: Mayor de tres números. Versión 2

- ▷ **Idea:** Ir descartando conforme se van comparando
- ▷ **Entradas y Salidas:** idem
- ▷ **Descripción:**

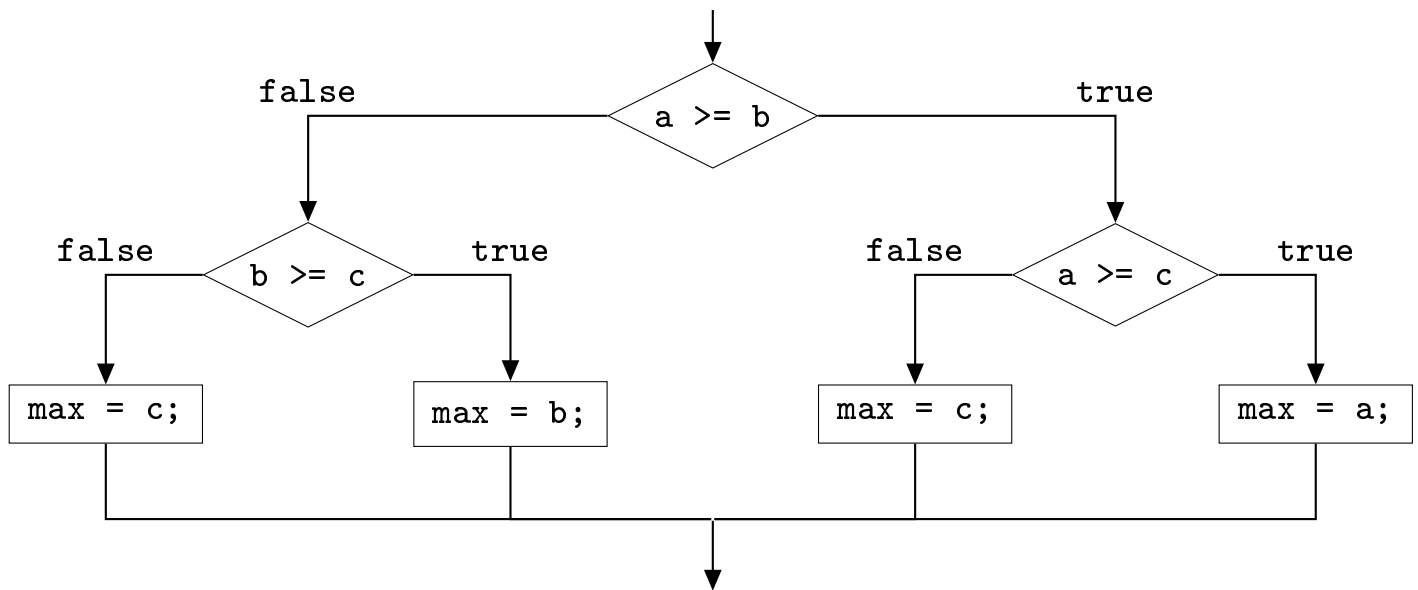
```
Si a es mayor que b, entonces
    Calcular el máximo entre a y c
En otro caso,
    Calcular el máximo entre b y c
```

- ▷ **Implementación:**

```
if (a >= b)
    if (a >= c)
        max = a;
    else
        max = c;
else
    if (b >= c)
        max = b;
    else
        max = c;
```



187



Inconvenientes:

- ▷ **Repetimos código:** `max = c;`
- ▷ **La solución es difícil de extender a más valores.** Observe cómo quedaría el mayor de cuatro números con esta aproximación:

```
if (a >= b)
    if (a >= c)
        if (a >= d)
            max = a;
        else
            max = d;
    else
        if (c >= d)
            max = c;
        else
            max = d;
else
    if (b >= c)
        if (b >= d)
            max = b;
        else
            max = d;
    else
        if (c >= d)
            max = c;
        else
            max = d;
```



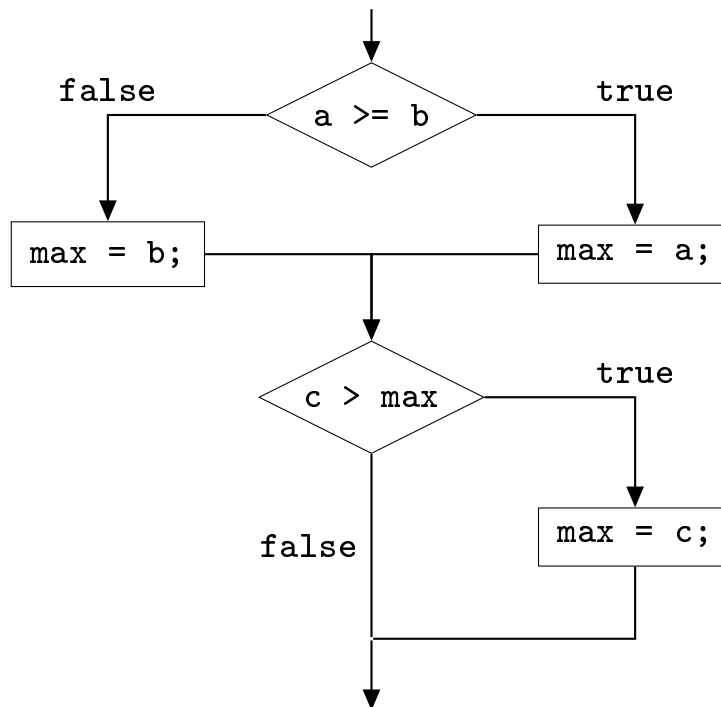
Ejemplo. El mayor de tres números (tercera aproximación)

Algoritmo: Mayor de tres números. Versión 3

- ▷ **Idea:** Ir actualizando el máximo.
- ▷ **Entradas y Salidas:** idem
- ▷ **Descripción e Implementación:**

```
/*  
Calcular el máximo (max) entre a y b.  
Calcular el máximo entre max y c.  
*/  
  
if (a >= b)  
    max = a;  
else  
    max = b;  
  
if (c > max)  
    max = c;
```

Observe que con el primer `if` garantizamos que la variable `max` tiene un valor (o bien `a` o bien `b`). Por tanto, en el último condicional basta usar una desigualdad estricta.



Ventajas:

- ▷ Es mucho más fácil de entender
- ▷ No repite código
- ▷ Es mucho más fácil de extender a varios valores. Veamos cómo quedaría con cuatro valores:

Algoritmo: Mayor de cuatro números

- ▷ **Entradas y Salidas:** idem
- ▷ **Descripción e Implementación:**

```
/*  
Calcular el máximo (max) entre a y b.  
Calcular el máximo entre max y c.  
Calcular el máximo entre max y d.  
*/  
  
if (a >= b)  
    max = a;  
else  
    max = b;  
  
if (c > max)  
    max = c;  
  
if (d > max)  
    max = d;
```



http://decsai.ugr.es/jccubero/FP/II_max.cpp

En general:

```
Calcular el máximo (max) entre a y b.  
Para cada uno de los valores restantes,  
    max = máximo entre max y el nuevo valor.
```

Diseña los algoritmos para que sean fácilmente extensibles a situaciones más generales.



Otra alternativa sería inicializar `max` al primer valor e ir comparando con el resto:

```
max = a;

if (b > max)
    max = b;

if (c > max)
    max = c;

if (d > max)
    max = d;
```

En general:

```
Inicializar el máximo a un valor cualquiera -a-
Para cada uno de los valores restantes,
    max = máximo entre max y el nuevo valor.
```

Algunas citas sobre la importancia de escribir código que sea fácil de entender:

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand".

Martin Fowler



"Programs must be written for people to read, and only incidentally for machines to execute".

Abelson & Sussman



"Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live".



Un principio de programación transversal:

Principio de Programación:

Sencillez (Simplicity)

Fomente siempre la sencillez y la legibilidad en la escritura de código



"There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."

C.A.R. Hoare



II.1.8.2. Descripción de un algoritmo

Se trata de describir la idea principal del algoritmo, de forma concisa y esquemática, sin entrar en detalles innecesarios.

```
if (a >= b)
    max = a;
else
    max = b;

if (c > max)
    max = c;
```

Una lamentable descripción del algoritmo:

Compruebo si $a \geq b$. En ese caso, lo que hago es asignarle a la variable max el valor a y si no le asigno el otro valor, b. Una vez hecho esto, paso a comprobar si el otro valor, es decir, c, es mayor que max (la variable anterior), en cuyo caso le asigno a max el valor c y si no, no hago nada.



El colmo:

Compruevo si $a \geq b$ y en ese caso lo que ago es asignarle a la variable max el valor a y sino le asigno el otro valor b una vez echo esto paso a comprobar si el otro valor es decir c es mayor que max (la variable anterior) en cuyo caso le asigno a max el valor c y sino no ago nada



▷ **Nunca parafrasearemos el código**

```
/* Si a es >= b, asignamos a max el valor a
   En otro caso, le asignamos b.
   Una vez hecho lo anterior, vemos si
   c es mayor que max, en cuyo caso
   le asignamos c
*/
```



```
if (a >= b)
    max = a;
else
    max = b;

if (c > max)
    max = c;
```

▷ **Seremos esquemáticos (pocas palabras en cada línea)**

```
/* Calcular el máximo entre a y b, y una vez hecho
   esto, pasamos a calcular el máximo entre el anterior,
   al que llamaremos max, y el nuevo valor c
*/
```



```
/* Calcular el máximo (max) entre a y b.
   Calcular el máximo entre max y c.
*/
if (a >= b)
    max = a;
else
    max = b;

if (c > max)
    max = c;
```



▷ **Comentaremos un bloque completo**

La descripción del algoritmo la incluiremos antes de un bloque, pero nunca entre las líneas del código. Esto nos permite separar las dos partes y poder leerlas independientemente.

```
// Calcular el máximo entre a y b
if (a >= b)
    max = a;
else
    // En otro caso:
    max = b;
// Calcular el máximo entre max y c
if (c > max)
    max = c;
```



Algunos autores incluyen la descripción a la derecha del código, pero es mucho más difícil de mantener ya que si incluimos líneas de código nuevas, se *descompone* todo el comentario:

```
if (a >= b)          // Calcular el máximo
    max = a;         // entre a y b
else
    max = b;

if (c > max)          // Calcular el máximo
    max = c;         // entre max y c
```



Use descripciones de algoritmos que sean CONCISAS con una presentación visual agradable y esquemática.

**No se hará ningún comentario de aquello que sea obvio.
Más vale poco y bueno que mucho y malo.**

Las descripciones serán de un BLOQUE completo.

Sólo se usarán comentarios al final de una línea en casos puntuales, para aclarar el código de dicha línea.



II.1.8.3. Descomposición de una solución en tareas

Si observamos que una misma operación se repite en todas las partes de una estructura condicional, debemos ver si es posible su extracción fuera de la estructura.

Ejemplo. Una empresa aplica la siguiente promoción en la venta de sus artículos: si el número de unidades pedidas es mayor de 50, se aplicará un descuento del 3 %. Lea el precio de venta por unidad, la cantidad de artículos pedida y calcule el precio final, aplicando un IVA del 16 %.

```
cin >> pvp_unidad >> unidades_vendidas;

if (unidades_vendidas < 50){
    pvp = pvp_unidad * unidades_vendidas;
    pvp = pvp * 1.16;
}
else{
    pvp = pvp_unidad * unidades_vendidas;
    pvp = pvp * 0.97;
    pvp = pvp * 1.16;
}
```



La sentencia `pvp = pvp_unidad * unidades_vendidas;` **se repite en ambas partes del condicional. Debemos sacarlo de éste y realizar la operación antes de entrar al if. La aplicación del IVA también se repite, por lo que debemos sacar dicha operación y realizarla después del condicional:**

```
cin >> pvp_unidad >> unidades_vendidas;
pvp = pvp_unidad * unidades_vendidas;

if (unidades_vendidas >= 50)
    pvp = pvp * 0.97;
pvp = pvp * 1.16;
```

Finalmente, como ya sabemos, no debemos utilizar *números mágicos* (vea la página 35) en el código sino constantes. Nos quedaría:

```
const int MIN_UNID_DSCTO = 50;
const double IV_DSCTO = 1 - 0.05;
const double IV_IVA = 1 + 0.16;
.....
cin >> pvp_unidad >> unidades_vendidas;

pvp = pvp_unidad * unidades_vendidas;

if (unidades_vendidas >= MIN_UNID_DSCTO)
    pvp = pvp * IV_DSCTO;

pvp = pvp * IV_IVA;
```



http://decsai.ugr.es/jccubero/FP/II_UnidadesVendidas.cpp

Lo que hemos hecho ha sido separar las tareas principales de este problema, a saber:

- ▷ Calcular el precio de todas las unidades vendidas
- ▷ Aplicar el descuento, en su caso
- ▷ Aplicar el IVA

Son tres tareas independientes que estaban duplicadas en la primera versión. A veces, la duplicación del código que se produce al repetir la resolución de una misma tarea en varios sitios no es tan fácil de identificar. Lo vemos en el siguiente ejemplo.

Ejemplo. Retomemos el ejemplo de la subida salarial de la página 166.

```
.....
salario_final = salario_base;

if (experiencia > 2) {
    if (edad >= 45 && salario_base < 1300){
        salario_final = salario_final * 1.04;

        if (numero_hijos > 2)
            salario_final = salario_final * 1.02;
    }
    else
        salario_final = salario_final * 1.01;
}
else
    salario_final = salario_final * 0.99;
```

Podemos resolver el problema descomponiendo la tarea principal en dos sub-tareas:

- 1. Calcular el porcentaje de actualización**
- 2. Aplicar dicho porcentaje**

```
double IV_salario;      // Índice de variación
.....
salario_final = salario_base;

// Calcular el porcentaje de actualización

if (experiencia > 2) {
    if (edad >= 45 && salario_base < 1300){
        IV_salario = 1.04;

        if (numero_hijos > 2)      // Observe esta asignación:
            IV_salario = IV_salario * 1.02;
    }
    else
        IV_salario = 1.01;
}
else
    IV_salario = 0.99;

// Aplicar dicho porcentaje

salario_final = salario_final * IV_salario;
```

Observe la actualización del 2%. Ahora queda mucho más claro que la subida del 2% es sobre la subida anterior (la del 4%)

Analice siempre con cuidado las tareas a resolver en un problema e intente separar los bloques de código que resuelven cada una de ellas.

IMPORTANT

Si tenemos previsto utilizar los límites de las subidas salariales y los porcentajes correspondientes en otros sitios del programa, debemos usar constantes en vez de literales:

```
const int MINIMO_EXPERIENCIA_ALTA = 2,
          MINIMO_FAMILIA_NUMEROSA = 2, ...

.....
const double IV_SENIOR_Y_SAL_BAJO      = 1.04,
              IV_NO_SENIOR_Y_SAL_BAJO = 1.01, ...

.....
es_experiencia_alta = experiencia > MINIMO_EXPERIENCIA_ALTA;
es_familia_numerosa = numero_hijos > MINIMO_FAMILIA_NUMEROSA;
es_salario_bajo      = salario_base < MAXIMO_SALARIO_BAJO;
es_edad_senior       = edad >= MINIMO_EDAD_SENIOR;

if (es_experiencia_alta){
    if (es_edad_senior && es_salario_bajo){
        IV_salario = IV_SENIOR_Y_SAL_BAJO;

        if (es_familia_numerosa)
            IV_salario = IV_salario * IV_FAMILIA_NUMEROSA;
    }
    else
        IV_salario = IV_NO_SENIOR_Y_SAL_BAJO;
}
else
    IV_salario = IV_SIN_EXPERIENCIA;

salario_final = salario_final * IV_salario;
.....
```



http://decsai.ugr.es/jccubero/FP/II_actualizacion_salarial.cpp

II.1.8.4. Las expresiones lógicas y el principio de una única vez

Según el principio de una única vez (página 112) no debemos repetir el código de una expresión en distintas partes del programa, si la evaluación de ésta no varía. Lo mismo se aplica si es una expresión lógica.

Ejemplo. Retomamos el ejemplo de subir la nota de la página 163. Imprimimos un mensaje específico si ha superado el examen escrito:

```
double nota_escrito;
.....
if (nota_escrito >= 4.5){
    nota_escrito = nota_escrito + 0.5;

    if (nota_escrito > 10)
        nota_escrito = 10;
}
.....
if (nota_escrito >= 4.5)                // <- repite código 😞
    cout << "Examen escrito superado con la nota: " << nota_escrito;
.....
```

Si ahora quisiéramos cambiar el criterio a que sea mayor estricto, tendríamos que modificar el código de dos líneas. Para resolver este problema, introducimos una variable lógica:

```
double nota_escrito;
bool escrito_superado;
.....
escrito_superado = nota_escrito >= 4.5;

if (escrito_superado){
    nota_escrito = nota_escrito + 0.5;

    if (nota_escrito > 10)
        nota_escrito = 10;
}
.....
if (escrito_superado)
    cout << "Examen escrito superado con la nota: " << nota_escrito;
.....
```



Nota:

Como siempre, deberíamos usar constantes

```
const double NOTA_MINIMA_APROBAR = 4.5;
const double MAX_NOTA = 10.0;
const double SUBIDA_NOTA_APROBADOS = 0.5;
```

en vez de los literales 4.5, 10, 0.5 en las sentencias del programa.

En resumen:

El uso de variables intermedias para determinar criterios nos ayuda a no repetir código y facilita la legibilidad de éste. Se les asigna un valor en un bloque y se observa su contenido en otro.

II.1.8.5. Separación de entradas/salidas y cálculos

En temas posteriores se introducirán herramientas para aislar bloques de código en módulos (clases, funciones, etc). Será importante que los módulos que hagan entradas y salidas de datos no realicen también otro tipo de cálculos ya que, de esta forma:

- ▷ Se separan responsabilidades.

Cada módulo puede actualizarse de forma independiente.

- ▷ Se favorece la reutilización entre plataformas (Linux, Windows, etc).

Las E/S en un entorno de ventanas como Windows no se hacen como se hacen en modo consola (con `cout`). Por lo tanto, el código que se encarga de esta parte será distinto. Pero la parte del código que se encarga de realizar los cálculos será el mismo: si dicho código está *empaquetado* en un módulo, éste podrá reutilizarse tanto en un programa que interactúe con la consola como en un programa que funcione en un entorno de ventanas.

Por ahora, trabajamos en un único fichero y sin módulos (éstos se introducen en el tema IV). Pero al menos, perseguiremos el objetivo de separar E/S y C, separando los bloques de código de cada parte.

La siguiente norma es un caso particular de la indicada en la página 197

Los bloques de código que realizan entradas o salidas de datos (`cin`, `cout`) estarán separados de los bloques que realizan cálculos.

IMPORTANT

El uso de variables intermedias nos ayuda a separar los bloques de E/S y C. Se les asigna un valor en un bloque y se observa su contenido en otro.

En la mayor parte de los ejemplos vistos en este tema hemos respetado esta separación. Veamos algunos ejemplos de lo que no debemos hacer.

Ejemplo. Retomamos los ejemplos de la página 139. Para no mezclar E/S y C, debemos sustituir el código siguiente:

```
// Cómputos mezclados con la salida de resultados:
```



```
if (entero % 2 == 0)
    cout << "\nEs par";
else
    cout << "\nEs impar";

cout << "\nFin del programa";
```

por:

```
bool es_par;
```

```
.....
```

```
// Cómputos:
```



```
es_par = entero % 2 == 0;
```

```
// Salida de resultados:
```

```
if (es_par)
    cout << "\nEs par";
else
    cout << "\nEs impar";

cout << "\nFin del programa";
```

Nos preguntamos si podríamos usar una variable de tipo `string` en vez de un `bool`:

```
string tipo_de_entero;
.....
// Cómputos:

if (entero % 2 == 0)
    tipo_de_entero = "es par"
else
    tipo_de_entero = "es impar";

// Salida de resultados:

if (tipo_de_entero == "es_par")
    cout << "\nEs par";
else
    cout << "\nEs impar";

cout << "\nFin del programa";
```



¿Localiza el error?

Estamos comparando con una cadena distinta de la asignada (cambia un único carácter)

Jamás usaremos un tipo `string` para detectar un número limitados de alternativas posibles ya que es propenso a errores.



Ejemplo. Retomamos el ejemplo del máximo de tres valores de la página 185

```
// Cómputos: Calculamos max

if (a >= b)
    max = a;
else
    max = b;

if (c > max)
    max = c;

// Salida de resultados: Observamos el valor de max

cout << "\nMáximo: " << max;
```



El siguiente código mezcla E/S y C dentro del mismo bloque condicional:

```
// Calculamos el máximo Y lo imprimimos en pantalla.
// Mezclamos E/S con C

if (a >= b)
    cout << "\nMáximo: " << a;
else
    cout << "\nMáximo: " << b;

if (c > max)
    cout << "\nMáximo: " << c;
```



Observe la similitud con lo visto en la página 196. En esta segunda versión del máximo no se han separado las tareas de calcular el máximo e imprimir el resultado.

Ejemplo. Retomamos el ejemplo de la edad y altura de una persona de la página 160. Para separar E/S y Cómputos, introducimos las variables intermedias `es_alto`, `es_mayor_edad`. Además, usamos constantes en vez de literales para representar y operar con los umbrales.

```
int main(){
    const int MAYORIA_EDAD = 18,
            UMBRAL_ALTURA_JOVENES = 175,
            UMBRAL_ALTURA_ADULTOS = 190;
    int edad, altura;
    bool es_alto, es_mayor_edad, umbral_altura;

    // Entrada de datos:

    cout << "Introduzca los valores de edad y altura: ";
    cin >> edad;
    cin >> altura;

    // Cómputos:

    es_mayor_edad = edad >= MAYORIA_EDAD;

    if (es_mayor_edad)
        umbral_altura = UMBRAL_ALTURA_ADULTOS;
    else
        umbral_altura = UMBRAL_ALTURA_JOVENES;

    es_alto = altura >= umbral_altura;
```



```
// Salida de resultados:

cout << "\n\n";

if (es_mayor_edad)
    cout << "Es mayor de edad";
else
    cout << "Es menor de edad";

if (es_alto)
    cout << "Es alto/a";
else
    cout << "No es alto/a";
}
```

http://decsai.ugr.es/jccubero/FP/II_altura.cpp

II.1.8.6. El tipo enumerado y los condicionales

Hay situaciones en las que necesitamos manejar información que sólo tiene unos cuantos valores posibles.

- ▷ **Calificación ECTS de un alumno:** {A, B, C, D, E, F, G}
- ▷ **Tipo de letra:** {es mayúscula, es minúscula, es otro carácter}
- ▷ **Puesto alcanzado en la competición:** {primero, segundo, tercero}
- ▷ **Día de la semana:** {lunes, ... , domingo}
- ▷ **Categoría laboral de un empleado:** {administrativo, programador, analista, directivo }
- ▷ **Tipo de ecuación de segundo grado:** {una única raíz, ninguna solución, ... }

Opciones con lo que conocemos hasta ahora:

- ▷ **Una variable de tipo `char` o `int`.** El inconveniente es que el código es propenso a errores:

```
char categoria_laboral;  
  
categoria_laboral = 'a';           // Analista  
categoria_laboral = 'm';           // adMinistrativo  
categoria_laboral = 'x';           // Categoría inexistente  
if (categoria_laboral == 'w')      // Categoría inexistente  
    .....
```



- ▷ **Un dato `bool` por cada categoría:**

```
bool es_administrativo, es_programador,  
     es_analista, es_directivo;
```



Inconvenientes: Debemos manejar cuatro variables por separado y además representan 8 opciones distintas, en vez de cuatro.

Solución: Usar un tipo enumerado. A un dato de tipo *enumerado* (*enumeration*) sólo se le puede asignar un número muy limitado de valores. Éstos son especificados por el programador. Primero se define el *tipo* (lo haremos antes del `main`) y luego la variable de dicho tipo.

Para nombrar los valores del enumerado, usaremos minúsculas. Es un estilo admitido en Google C++ Style Guide, aunque recomiendan ir sustituyéndolo por el uso de mayúsculas (como se hace con las constantes) Nosotros usaremos minúsculas para asemejarlo a los valores `true`, `false` de un `bool`. Usaremos el estilo `UpperCamelCase` para el nombre del tipo de dato.

```
#include <iostream>
using namespace std;

enum class CalificacionECTS
    {A, B, C, D, E, F, G}; // No van entre comillas!
enum class PuestoPodium
    {primero, segundo, tercero};
enum class CategoriaLaboral
    {administrativo, programador, analista, directivo};

int main(){
    CalificacionECTS nota;
    PuestoPodium      podium;
    CategoriaLaboral  categoria_laboral;

    nota              = CalificacionECTS::A;
    podium            = PuestoPodium::primero;
    categoria_laboral = CategoriaLaboral::programador;

    // Las siguientes sentencias dan error de compilación
    // categoria_laboral = CategoriaLaboral::peon
```




```
// categoria_laboral = peon;  
// categoria_laboral = 'a';  
// cin >> categoria_laboral;  
.....  
}
```

El tipo enumerado puede verse como una extensión del tipo `bool` ya que nos permite manejar más de dos opciones excluyentes.

Al igual que un `bool`, un dato enumerado contendrá un único valor en cada momento. La diferencia está en que con un `bool` sólo tenemos 2 posibilidades y con un enumerado tenemos más (pero no tantas como las 256 de un `char`, por ejemplo).

¿Qué operaciones se pueden hacer sobre un enumerado? Por ahora, sólo tiene sentido la comparación de igualdad.

Ejemplo. Modifique el código del programa que calcula las raíces de una ecuación de segundo grado (página 162) para separar las E/S de los cálculos. Debemos introducir una variable de tipo enumerado que nos indique el tipo de ecuación (una única raíz doble, dos raíces reales, etc)

```
#include <iostream>
#include <cmath>
using namespace std;

enum class TipoEcuacion
{
    una_raiz_doble, dos_raices_reales, ninguna_raiz_real,
    recta_con_una_raiz, no_es_ecuacion};

int main(){
    int a, b, c;
    int denominador;
    double radical, radicando, raiz1, raiz2;
    TipoEcuacion tipo_ecuacion;

    // Entrada de datos:

    cout << "\nIntroduce coeficiente de segundo grado: ";
    cin >> a;
    cout << "\nIntroduce coeficiente de 1er grado: ";
    cin >> b;
    cout << "\nIntroduce coeficiente independiente: ";
    cin >> c;

    // Cálculos:

    if (a != 0) {
        denominador = 2*a;
        radicando = b*b - 4*a*c;
```



```
    if (radicando == 0){
        raiz1 = -b / denominador;
        tipo_ecuacion = TipoEcuacion::una_raiz_doble;
    }
    else{
        if (radicando > 0){
            radical = sqrt(radicando);
            raiz1 = (-b + radical) / denominador;
            raiz2 = (-b - radical) / denominador;
            tipo_ecuacion = TipoEcuacion::dos_raices_reales;
        }
        else
            tipo_ecuacion = TipoEcuacion::ninguna_raiz_real;
    }
}
else{
    if (b != 0){
        raiz1 = -c / b;
        tipo_ecuacion = TipoEcuacion::recta_con_una_raiz;
    }
    else
        tipo_ecuacion = TipoEcuacion::no_es_ecuacion;
}

// Salida de Resultados:

cout << "\n\n";

if (tipo_ecuacion == TipoEcuacion::una_raiz_doble)
    cout << "Sólo hay una raíz doble: " << raiz1;
else if (tipo_ecuacion == TipoEcuacion::dos_raices_reales)
    cout << "Las raíces son: " << raiz1 << " y " << raiz2;
else if (tipo_ecuacion == TipoEcuacion::ninguna_raiz_real)
```

```
    cout << "No hay raíces reales.";
else if (tipo_ecuacion == TipoEcuacion::recta_con_una_raiz)
    cout << "Es una recta. La única raíz es: " << raiz1;
else if (tipo_ecuacion == TipoEcuacion::no_es_ecuacion)
    cout << "No es una ecuación.";
```

http://decsai.ugr.es/jccubero/FP/II_ecuacion_segundo_grado.cpp

Si se prefiere, puede usarse una estructura condicional múltiple, pero recordemos que es una estructura a evitar ya que nos obliga a tener que incluir explícitamente un `break` en cada `case`.

```
switch (tipo_ecuacion){
    case TipoEcuacion::una_raiz_doble:
        cout << "Sólo hay una raíz doble: " << raiz1;
        break;
    case TipoEcuacion::dos_raices_reales:
        cout << "Las raíces son: " << raiz1 << " y " << raiz2;
        break;
    case TipoEcuacion::ninguna_raiz_real:
        cout << "No hay raíces reales.";
        break;
    case TipoEcuacion::recta_con_una_raiz:
        cout << "Es una recta. La única raíz es: " << raiz1;
        break;
    case TipoEcuacion::no_es_ecuacion:
        cout << "No es una ecuación.";
        break;
}
```

Ampliación:



Observe que la lectura con `cin` de un enumerado (`cin >> categoria_laboral;`) produce un error en tiempo de ejecución. ¿Cómo leemos entonces los valores de un enumerado desde un dispositivo externo? Habrá que usar una codificación (con caracteres, enteros, etc) y traducirla al enumerado correspondiente.

```
enum class CategoriaLaboral
    {administrativo, programador, analista, directivo};
int main(){
    CategoriaLaboral categoria_laboral;
    char char_categoria_laboral;
    .....
    cin >> char_categoria_laboral;

    if (char_categoria_laboral == 'm')
        categoria_laboral = CategoriaLaboral::administrativo
    else if (char_categoria_laboral == 'p')
        categoria_laboral = CategoriaLaboral::programador
    else if .....
    .....
    if (char_categoria_laboral == CategoriaLaboral::administrativo)
        retencion_fiscal = RETENCION_BAJA;
    else if (char_categoria_laboral == CategoriaLaboral::programador)
        retencion_fiscal = RETENCION_MEDIA;

    salario_netto = salario_bruto -
        salario_bruto * retencion_fiscal/100.0;
```

Una vez leídos los datos y hecha la transformación al enumerado correspondiente, nunca más volveremos a usar los caracteres sino la variable de tipo enumerado.

II.2. Estructuras repetitivas

Una *estructura repetitiva (iteration/loop)* (también conocidas como *bucles, ciclos o lazos*) permite la ejecución de una secuencia de sentencias:

- ▷ o bien, hasta que se satisface una determinada condición → *Bucle controlado por condición (Condition-controlled loop)*
- ▷ o bien, un número determinado de veces → *Bucle controlado por contador (Counter controlled loop)*

II.2.1. Bucles controlados por condición: pre-test y post-test

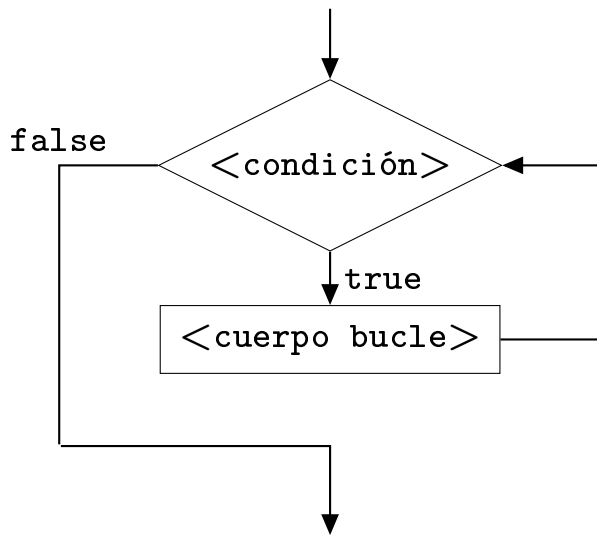
II.2.1.1. Formato

Pre-test	Post-test
<pre>while (<condición>) { <cuerpo bucle> }</pre>	<pre>do{ <cuerpo bucle> }while (<condición>);</pre>

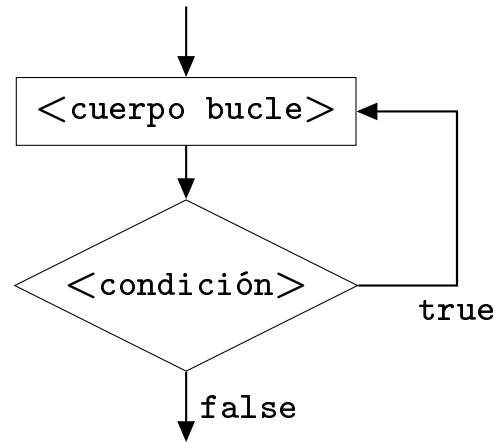
Funcionamiento: En ambos, se va ejecutando el cuerpo del bucle mientras la condición sea verdad.

- ▷ En un *bucle pre-test (pre-test loop)* (`while`) se evalúa la condición antes de entrar al bucle y luego (en su caso) se ejecuta el cuerpo.
- ▷ En un *bucle post-test (post-test loop)* (`do while`) primero se ejecuta el cuerpo y luego se evalúa la condición.

Cada vez que se ejecuta el cuerpo del bucle diremos que se ha producido



(a) while pre test



(b) do-while post test

una *iteración (iteration)*

Al igual que ocurre en la estructura condicional, si los bloques de instrucciones contienen más de una línea, debemos englobarlos entre llaves. En el caso del post-test se recomienda usar el siguiente estilo:

```
do{  
    <cuerpo bucle>  
}while (<condición>);
```



en vez de:

```
do{  
    <cuerpo bucle>  
}  
while (<condición>);
```



ya que, en la segunda versión, si no vemos las instrucciones antes del while, da la impresión que éste es un pre-test.

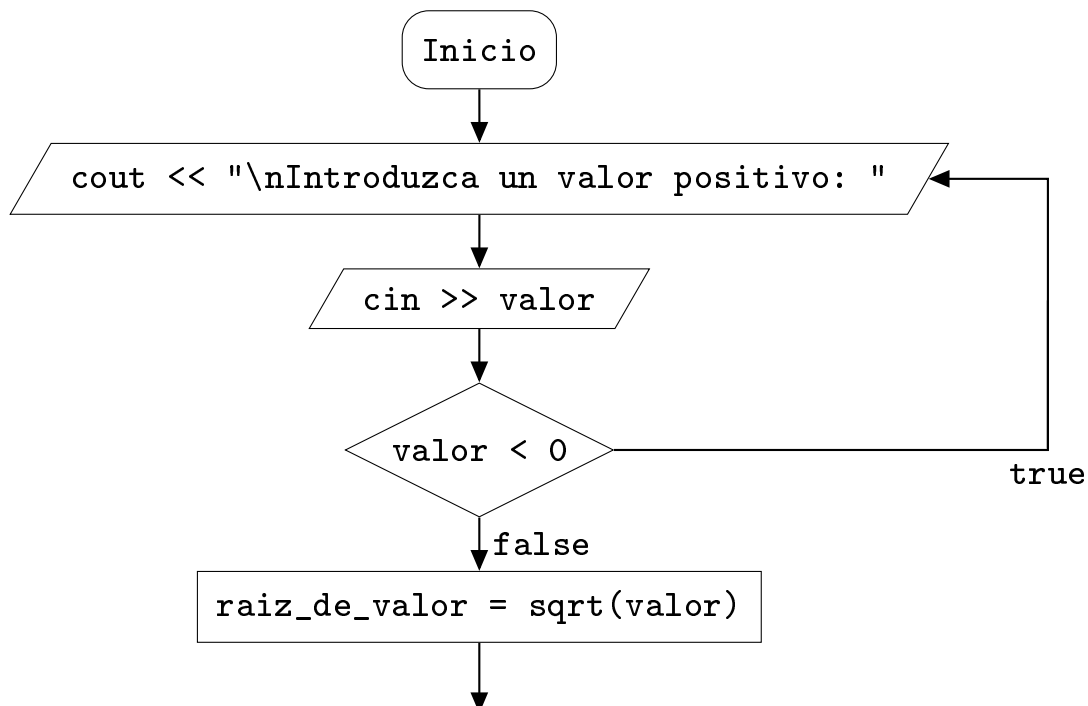
II.2.1.2. Algunos usos de los bucles

Ejemplo. Cree un *filtro (filter)* de entrada de datos: Leer un valor y no permitir al usuario que lo introduzca fuera de un rango determinado. Por ejemplo, que sea un entero positivo para poder calcular la raíz cuadrada:

```
int main(){
    double valor;
    double raiz_de_valor;

    do{
        cout << "\nIntroduzca un valor positivo: ";
        cin >> valor;
    }while (valor < 0);

    raiz_de_valor = sqrt(valor);
    .....
}
```



Nota:

El filtro anterior no nos evita todos los posibles errores. Por ejemplo, si se introduce un valor demasiado grande, se produce un desbordamiento y el resultado almacenado en `valor` es indeterminado.

Nota:

Observe que el estilo de codificación se rige por las mismas normas que las indicadas en la estructura condicional (página 131)

Ejemplo. Escriba 20 líneas con 5 estrellas cada una.

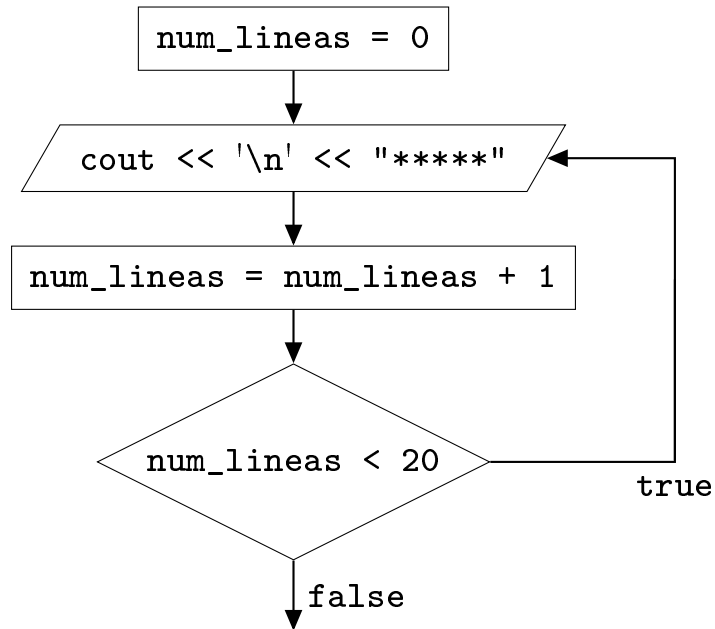
Necesitamos una variable `num_lineas` para contar el número de líneas impresas. Vamos a ver distintas versiones para resolver este problema:

a) Post-test con condición \leq

```
.....  
int num_lineas;  
  
num_lineas = 1;  
  
do{  
    cout << '\n' << "*****" ;  
    num_lineas = num_lineas + 1;    // nuevo = antiguo + 1  
}while (num_lineas <= 20);
```

b) Post-test con condición $<$

```
num_lineas = 0;  
  
do{  
    cout << '\n' << "*****" ;  
    num_lineas = num_lineas + 1;  
}while (num_lineas < 20);
```



c) Pre-test con condición <=

```
num_lineas = 1;

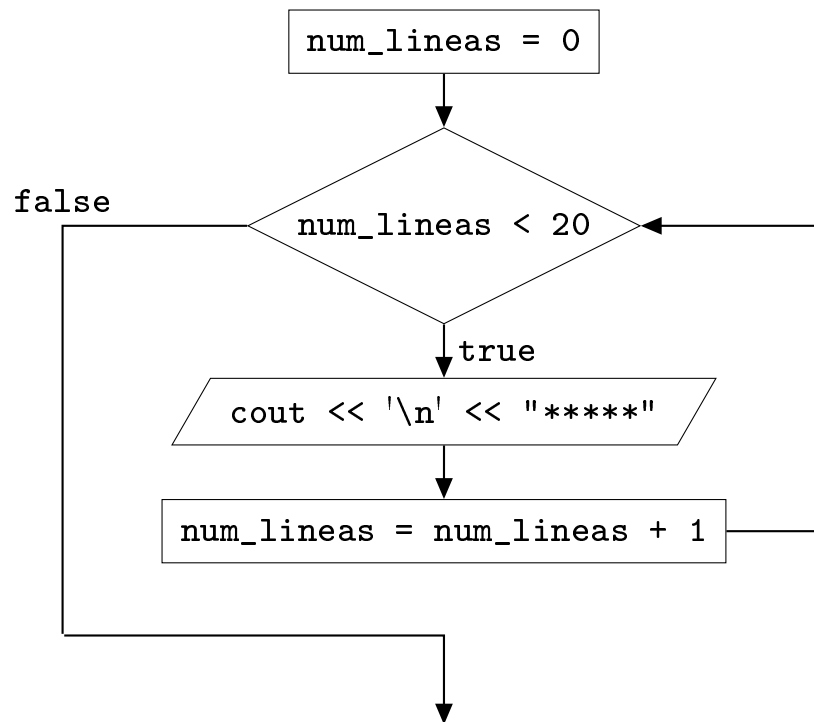
while (num_lineas <= 20){
    cout << '\n' << "*****" ;
    num_lineas = num_lineas + 1;
}
```

d) Pre-test con condición <

```
num_lineas = 0;    // Llevo 0 líneas impresas

while (num_lineas < 20){
    cout << '\n' << "*****" ;
    num_lineas = num_lineas + 1;
}
```





La opción d) es la preferible ya que es más fácil de entender: dentro del bucle, **justo antes** de comprobar la condición, la variable `total` contiene el número de líneas que han sido impresas. Además, una vez que termina el bucle la variable sigue valiendo lo mismo (ni uno más, ni uno menos). Finalmente, la inicialización a cero es coherente ya que al inicio aún no se ha impreso ninguna línea.

Nota:

Podríamos usar el operador de incremento:

```
while (num_lineas < 20){
    cout << '\\n' << "*****" ;
    num_lineas++;
}
```

¿Cuándo elegiremos un bucle pre-test y cuándo un post-test? Para ello, debemos responder a la siguiente pregunta: ¿Hay situaciones en las que no queremos que se ejecute el cuerpo del bucle?

Sí \Rightarrow pre-test

No \Rightarrow post-test

Ejemplo. Lea un número positivo `tope` e imprima `tope` líneas con 5 estrellas cada una.

```
cout << "\n¿Cuántas líneas de asteriscos quiere imprimir? ";

do{
    cin >> tope;
}while (tope < 0);    // Sale del filtro con un valor >= 0

num_lineas = 0;

do{
    cout << '\n' << "*****" ;
    num_lineas++;
}while (num_lineas < tope);
```

Problema: ¿Qué ocurre si `tope` vale 0?

Ejercicio. Resuelva el problema anterior con un bucle pre-test

Consejo: *Fomente el uso de los bucles pre-test. Lo usual es que haya algún caso en el que no queramos ejecutar el cuerpo del bucle ni siquiera una vez.*



Ejemplo. Sume los 100 primeros números positivos.

```
int suma, valor;  
  
valor = 1;  
  
while (valor <= 100){  
    suma = suma + valor;  
    valor++;  
}
```

En cada iteración, la instrucción

```
suma = suma + valor;
```

va acumulando en la variable `suma` las sumas anteriores.

Y la instrucción

```
valor++;
```

va sumándole 1 a la variable `valor`.

valor	suma	
1	1	= 1
2	1 + 2	= 3
3	3 + 3	= 6
4	6 + 4	= 10
...

Ejercicio. Calcule el número de dígitos que tiene un entero

57102 -> 5 dígitos

45 -> 2 dígitos

Algoritmo: Número de dígitos de un entero.

▷ **Entradas:** n

▷ **Salidas:** num_digitos

▷ **Descripción e implementación:**

Ir dividiendo n por 10 hasta llegar a una cifra

El número de dígitos será el número de iteraciones

http://decsai.ugr.es/jccubero/FP/II_numero_de_digitos.cpp

Ejemplo. Lea un entero `tope` positivo y escriba los pares \leq `tope`. ¿Cuál es el problema de esta solución?:

```
do{
    cin >> tope
}while (tope < 0);

par = 0;                // Primer candidato

while (par <= tope){
    par = par + 2;
    cout << par << " ";
}
```

Al final, escribe uno más. Cambiamos el orden de las instrucciones:

```
do{
    cin >> tope
}while (tope < 0);

par = 0;                // Primer candidato

while (par <= tope){    // ¿Es bueno?
    cout << par;        // Si => Imprímelo
    par = par + 2;      //      Calcular nuevo candidato
}                      // No => Salir
```

Si no queremos que salga 0, cambiaríamos la inicialización:

```
par = 2;                // Primer candidato
```

En el diseño de los bucles siempre hay que comprobar el correcto funcionamiento en los casos extremos (primera y última iteración)

II.2.1.3. Bucles para lectura de datos

En muchas ocasiones leeremos datos desde un dispositivo y tendremos que controlar una condición de parada. Habrá que controlar especialmente el primer y último valor leído.

Ejemplo. Realice un programa que sume una serie de valores leídos desde teclado, hasta que se lea el valor -1 (terminador = -1)

```
#include <iostream>
using namespace std;

int main(){
    const int TERMINADOR = -1;
    int suma, numero;

    suma = 0;

    do{
        cin >> numero;
        suma = suma + numero;
    }while (numero != TERMINADOR);

    cout << "\nLa suma es " << suma;
}
```

Caso problemático: El último. Procesa el -1 y lo suma.

Una primera solución:

```
do{  
    cin >> numero;  
  
    if (numero != TERMINADOR)  
        suma = suma + numero;  
}while (numero != TERMINADOR);
```



Funciona, pero evalúa dos veces la misma condición, lo cual es ineficiente y, mucho peor, duplica código al repetir la expresión `numero != TERMINADOR`.

Soluciones:

- ▷ Usar variables lógicas.
- ▷ Técnica de *lectura anticipada* . Leemos el primer valor antes de entrar al bucle y comprobamos si hay que procesarlo (el primer valor podría ser ya el terminador)

Solución con una variable lógica:

```
bool seguir_leyendo;  
suma = 0;  
  
do{  
    cin >> numero;  
  
    seguir_leyendo = (numero != TERMINADOR);  
  
    if (seguir_leyendo)          // Comprobamos si es el terminador  
        suma = suma + numero;    // Lo procesamos  
}while (seguir_leyendo);
```



La expresión que controla la condición de parada (`numero != TERMINADOR`); sólo aparece en un único sitio, por lo que si cambiase el criterio de parada, sólo habría que cambiar el código en dicho sitio.


Es verdad que repetimos la observación de la variable `seguir_leyendo` (en el `if` y en el `while`) pero no repetimos la evaluación de la expresión anterior.


Solución con lectura anticipada:

```
suma = 0;
cin >> numero;

while (numero != TERMINADOR) {
    suma = suma + numero;
    cin >> numero;
}

cout << "\nLa suma es " << suma;
```

 // Lectura anticipada del
// primer candidato

 // Comprobamos si es el terminador
// Lo procesamos
// Leemos siguiente candidato

http://decsai.ugr.es/jccubero/FP/II_suma_lectura_anticipada.cpp

A tener en cuenta:

- ▷ La primera vez que entra al bucle, la instrucción

```
while (numero != TERMINADOR)
```

hace las veces de un condicional. De esta forma, controlamos si hay que procesar o no el primer valor.

- ▷ Si el primer valor es el terminador, el algoritmo funciona correctamente.
- ▷ Hay cierto código repetido `cin >> numero;`, pero es aceptable. Mucho peor es repetir la expresión `numero != TERMINADOR` ya que es mucho más fácil que pueda cambiar en el futuro (porque cambie el criterio de parada)
- ▷ Dentro de la misma estructura repetitiva estamos mezclando las entradas (`cin >> numero;`) de datos con los cálculos (`suma = suma + numero;`), violando lo visto en la página 201. Por ahora no podemos evitarlo, ya que necesitaríamos almacenar los valores en un dato compuesto, para luego procesarlo. Lo resolveremos con el uso de vectores en el tema III.

Para no repetir código en los bucles que leen datos, normalmente usaremos la técnica de lectura anticipada:

```
cin >> dato;

while (dato no es último){
    procesar_dato
    cin >> dato;
}
```

A veces también puede ser útil introducir variables lógicas que controlen la condición de terminación de lectura:

```
do{
    cin >> dato;

    test_dato = ¿es el último?

    if (test_dato)
        procesar_dato
}while (test_dato);
```

Ejercicio. Lea enteros hasta llegar al cero. Imprima el número de pares e impares leídos.

Ejemplo. Retome el ejemplo de la subida salarial de la página 198. Creamos un programa para leer los datos de muchos empleados. El primer dato a leer será la experiencia. Si es igual a -1, el programa terminará.

```
const int TERMINADOR = -1;
.....
cin >> experiencia;

while (experiencia != TERMINADOR){
    cin >> salario_base; // Suponemos que el salario base es
                        // distinto en cada empleado

    cin >> edad;
    cin >> numero_hijos;
    .....
    if (es_aplicable_subida){
        .....
    }

    cin >> experiencia;
}
```

http://decsai.ugr.es/jccubero/FP/II_actualizacion_salarial_lectura_datos.cpp

Ejemplo. Lea datos enteros desde teclado hasta que se introduzca el 0. Calcule el número de valores introducidos y el mínimos de todos ellos. Vamos a ir mejorando la solución propuesta.

```
/*
Algoritmo:
    min contendrá el mínimo hasta ese momento

    Leer datos hasta llegar al terminador
    Actualizar el contador de valores introducidos
    Actualizar, en su caso, min
*/

cin >> dato;

while (dato != TERMINADOR){
    validos_introducidos++;

    if (dato < min)
        min = dato;

    cin >> dato;
}
```

¿Qué valor le damos a min la primera vez? ¿El mayor posible para garantizar que la expresión `dato < min` sea true la primera vez?

```
cin >> dato;
min = 32768;

while (dato != TERMINADOR){
    validos_introducidos++;

    if (dato < min)
```



```
    min = dato;  
  
    cin >> dato;  
}
```

¿Y si el compilador usa 32 bits en vez de 16 bits para representar un `int`?

¿Y si nos equivocamos al especificar el literal?

Recuerde lo visto en la página 35: evite siempre el uso de *números mágicos*.

Para resolver este problema, podríamos asignarle un valor dentro del bucle, detectando la primera iteración:

```
bool es_primera_vez;  
.....  
es_primera_vez = true;  
cin >> dato;  
  
while (dato != TERMINADOR){  
    validos_introducidos++;  
  
    if (es_primera_vez){  
        min = dato;  
        es_primera_vez = false;  
    }  
    else{  
        if (dato < min)  
            min = dato;  
    }  
  
    cin >> dato;  
}
```



Evite, en la medida de lo posible, preguntar por algo que sabemos de antemano que sólo va a ser verdadero en la primera iteración.

Realmente, la solución a nuestro problema es muy sencilla. Basta inicializar min al primer valor leído:

Algoritmo: Mínimo de varios valores.

- ▷ **Entradas:** Enteros hasta introducir un terminador
Salidas: El mínimo de ellos y el número de valores introducidos

- ▷ **Descripción e Implementación:**

```
/*  
Algoritmo:  
    Usamos una variable min, que contendrá el  
    mínimo hasta ese momento  
    Leer primer dato e inicializar min a dicho valor.  
    Leer datos hasta llegar al terminador  
        Actualizar el contador de valores introducidos  
        Actualizar, en su caso, min  
*/  
cin >> dato;  
min = dato;  
validos_introducidos = 0;  
  
while (dato != TERMINADOR){  
    validos_introducidos++;  
  
    if (dato < min)  
        min = dato;  
  
    cin >> dato;  
}
```

http://decsai.ugr.es/jccubero/FP/II_min_hasta_terminador.cpp

Nota:

Si se necesita saber el máximo valor entero representable, se puede recurrir a la función `numeric_limits<int>::max()`: de la biblioteca `limits`. En el ejemplo anterior, en vez de usar el número mágico `32768`, podríamos poner lo siguiente:

```
#include <limits>

.....

    cin >> dato;
    min = numeric_limits<int>::max();

    while (dato != TERMINADOR){
        validos_introducidos++;

        if (dato < min)
            min = dato;

        cin >> dato;
    }
```

Esta solución sí sería correcta. En cualquier caso, en el ejemplo que nos ocupa, la solución que hemos propuesto con la inicialización de `min` al primer dato leído es mucho más clara.

II.2.1.4. Bucles sin fin

Ejemplo. Imprima los divisores de un valor.

```
int divisor, valor, ultimo_divisor_posible;

cin >> valor;
ultimo_divisor_posible = valor / 2;
divisor = 2;

while (divisor <= ultimo_divisor_posible){
    if (valor % divisor == 0)
        cout << "\n" << divisor << " es un divisor de " << valor;

    divisor++;
}
```

¿Qué pasa si introducimos un else?

```
while (divisor <= ultimo_divisor_posible){
    if (valor % divisor == 0)
        cout << "\n" << divisor << " es un divisor de " << valor;
    else
        divisor++;
}
```

Es un bucle sin fin.

Debemos garantizar que en algún momento, la condición del bucle se hace falsa.

Tenga especial cuidado dentro del bucle con los condicionales que modifican alguna variable presente en la condición.

Ejemplo. ¿Cuántas iteraciones se producen?

```
contador = 1;

while (contador != 10) {
    contador = contador + 2;
}
```

Llega al máximo entero representable (2147483647). Al sumar 2, se produce un desbordamiento, obteniendo -2147483647. Al ser impar nunca llegará a 10, por lo que se producirá de nuevo la misma situación y el bucle no terminará.

Solución. Fomente el uso de condiciones de desigualdad:

```
contador = 1;

while (contador <= 10) {
    contador = contador + 2;
}
```

II.2.1.5. Condiciones compuestas

Es normal que necesitemos comprobar más de una condición en un bucle. Dependiendo del algoritmo necesitaremos conectarlas con `&&` o con `||`.

Ejemplo. Lea una opción de un menú. Sólo se admite s ó n.

```
char opcion;
do{
    cout << "¿Desea formatear el disco?";
    cin >> opcion;
}while ( opcion != 's'    opcion != 'S'
        opcion != 'n'    opcion != 'N' );
```

¿Cuándo quiero salir del bucle? Cuando *cualquiera* de las condiciones sea false. ¿Cual es el operador que cumple lo siguiente?:

```
false  Operador  <lo que sea>  =  false
true   Operador  true           =  true
```

Es el operador `&&`

Mejor aún si pasamos previamente el carácter a mayúscula y nos ahorramos dos condiciones:

```
do{
    cout << "Desea formatear el disco";
    cin >> opcion;
    opcion = toupper(opcion);
}while ( opcion != 'S' && opcion != 'N' );
```

Ejemplo. Calcule el máximo común divisor de dos números a y b.

Algoritmo: Máximo común divisor.

▷ **Entradas:** Los dos enteros a y b

Salidas: el entero `max_com_div`, máximo común divisor de a y b

▷ **Descripción e Implementación:**

```
/* Primer posible divisor = el menor de ambos
   Mientras divisor no divida a ambos,
   probar con el anterior */

if (b < a)
    menor = b;
else
    menor = a;

divisor = menor;

while (a % divisor != 0      b % divisor != 0)
    divisor --;

max_com_div = divisor;
```

¿Cuándo quiero salir del bucle? Cuando ambas condiciones, **simultáneamente**, sean false. En cualquier otro caso, entro de nuevo al bucle.

¿Cual es el operador que cumple lo siguiente?:

```
true   Operador <lo que sea> = true
false  Operador false  = false
```

Es el operador ||

```
while (a % divisor != 0 || b % divisor != 0)
    divisor --;

max_com_div = divisor;
```

En la construcción de bucles con condiciones compuestas, empiece planteando las condiciones simples que la forman. Piense cuándo queremos salir del bucle y conecte adecuadamente dichas condiciones simples, dependiendo de si nos queremos salir cuando todas simultáneamente sean false (conectamos con ||) o cuando cualquiera de ellas sea false (conectamos con &&)

Ejercicio. ¿Qué pasaría si a y b son primos relativos?

El uso de variables lógicas hará que las condiciones sean más fáciles de entender:

```
bool mcd_encontrado;  
.....  
mcd_encontrado = false;  
  
while (!mcd_encontrado){  
    if (a % divisor == 0 && b % divisor == 0)  
        mcd_encontrado = true;  
    else  
        divisor--;  
}  
  
max_com_div = divisor;
```

http://decsai.ugr.es/jccubero/FP/II_maximo_comun_divisor.cpp

Ambas soluciones son equivalentes. Formalmente:

```
a % divisor != 0 || b % divisor != 0  
equivale a  
!(a % divisor == 0 && b % divisor == 0)
```

Diseñe las condiciones compuestas de forma que sean fáciles de leer: en muchas situaciones, nos ayudará introducir variables lógicas a las que se les asignará un valor para salir del bucle cuando se verifique cierta condición de parada.

Ejercicio. ¿Qué pasaría si quitásemos el `else`?

II.2.1.6. Bucles que buscan

Una tarea típica en programación es buscar un valor. Si sólo estamos interesados en buscar uno, tendremos que salir del bucle en cuanto lo encontremos y así aumentar la eficiencia.

Ejemplo. Compruebe si un número entero positivo es primo. Para ello, debemos buscar un divisor suyo. Si lo encontramos, no es primo.

```
int valor, divisor, ultimo_divisor_posible;
bool es_primo;

cout << "Introduzca un numero natural: ";
cin >> valor;

es_primo = true;
divisor = 2;
ultimo_divisor_posible = divisor / 2;

while (divisor <= ultimo_divisor_posible){
    if (valor % divisor == 0)
        es_primo = false;
    divisor++;
}

if (es_primo)
    cout << valor << " es primo\n";
else{
    cout << valor << " no es primo\n";
    cout << "\nSu primer divisor es: " << divisor;
}
```



Funciona pero es ineficiente. Nos debemos salir del bucle en cuanto sepamos que no es primo. Usamos la variable `es_primo`.

```
int main(){
    int valor, divisor, ultimo_divisor_posible;
    bool es_primo;

    cout << "Introduzca un numero natural: ";
    cin >> valor;

    es_primo = true;
    divisor = 2;
    ultimo_divisor_posible = valor / 2;

    while (divisor <= ultimo_divisor_posible && es_primo){
        if (valor % divisor == 0)
            es_primo = false;
        else
            divisor++;
    }

    if (es_primo)
        cout << valor << " es primo";
    else{
        cout << valor << " no es primo";
        cout << "\nSu primer divisor es: " << divisor;
    }
}
```



http://decsai.ugr.es/jccubero/FP/II_primo.cpp

Los algoritmos que realizan una búsqueda, deben salir de ésta en cuanto se haya encontrado el valor. Normalmente, usaremos una variable lógica para controlarlo.

IMPORTANT

Nota:

Al usar `else` garantizamos que al salir del bucle, la variable `divisor` contiene el primer divisor de `valor`

Ampliación:

Incluso podríamos quedarnos en `sqrt(valor)`, ya que si `valor` no es primo, tiene al menos un divisor menor que `sqrt(valor)`. En cualquier caso, `sqrt` es una operación costosa y habría que evaluar la posible ventaja en su uso.

```
es_primo = true;
ultimo_divisor_posible = sqrt(1.0 * valor);
                        // Necesario forzar casting a double
divisor = 2;

while (divisor <= ultimo_divisor_posible && es_primo){
    if (valor % divisor == 0)
        es_primo = false;
    else
        divisor++;
}
```



II.2.2. Programando como profesionales

II.2.2.1. Evaluación de expresiones dentro y fuera del bucle

En ocasiones, una vez terminado un bucle, es necesario comprobar cuál fue la condición que hizo que éste terminase. Veamos cómo hacerlo correctamente.

Ejemplo. Desde un sensor se toman datos de la frecuencia cardíaca de una persona. Emita una alarma cuando se encuentren fuera del rango [45, 120] indicando si es baja o alta. El sensor emite el valor -1 si la batería es insuficiente.

```
int main(){
    const int MIN_LATIDOS = 45;
    const int MAX_LATIDOS = 120;
    const int SIN_BATERIA = -1;
    int latidos;

    // La siguiente versión repite código:

    do{
        cin >> latidos;
    }while (MIN_LATIDOS <= latidos && latidos <= MAX_LATIDOS &&
           latidos != SIN_BATERIA);

    if (latidos == SIN_BATERIA)
        cout << "\nBatería baja";
    else if (latidos < MIN_LATIDOS || latidos > MAX_LATIDOS)
        cout << "\nNúmero de latidos anormal: " << latidos;
    else
        cout << "\nError desconocido";
}
```



Se repite código ya que, por ejemplo, la expresión `latidos < MIN_LATIDOS` equivale a `!(MIN_LATIDOS <= latidos)` Para resolverlo, introducimos variables lógicas intermedias:

```
int main(){
    .....
    bool frecuencia_cardiaca_anormal, sensor_sin_bateria;

    do{
        cin >> latidos;
        frecuencia_cardiaca_anormal = latidos < MIN_LATIDOS
                                   ||
                                   latidos > MAX_LATIDOS;
        sensor_sin_bateria = latidos == SIN_BATERIA;
    }while (!frecuencia_cardiaca_anormal && !sensor_sin_bateria);

    if (sensor_sin_bateria)
        cout << "\nBatería baja";
    else if (frecuencia_cardiaca_anormal)
        cout << "\nNúmero de latidos anormal: " << latidos;
    else
        cout << "\nError desconocido";
}
```



Y mejor aún, podríamos usar un enumerado para distinguir todas las situaciones posibles (frecuencia normal, anormalmente baja y anormalmente alta):

http://decsai.ugr.es/jccubero/FP/II_frecuencia_cardiaca.cpp

Si una vez que termina un bucle controlado por varias condiciones, necesitamos saber cuál de ellas hizo que terminase éste, introduciremos variables intermedias para determinarlo. Nunca repetiremos la evaluación de condiciones.

II.2.2.2. Bucles que no terminan todas sus tareas

Ejemplo. Queremos leer las notas de un alumno y calcular la media aritmética. El alumno tendrá un máximo de cuatro calificaciones. Si tiene menos de cuatro, introduciremos cualquier negativo para indicarlo.

```
suma = 0;
cin >> nota;
total_introducidos = 1;

while (nota >= 0 && total_introducidos <= 4){
    suma = suma + nota;
    cin >> nota;
    total_introducidos++;
}
media = suma/total_introducidos;
```



Problema: Lee la quinta nota y se sale, pero ha tenido que leer dicho valor.

Solución: O bien cambiamos la inicialización de `total_introducidos` a 0, o bien leemos hasta menor estricto de 4; pero entonces, el cuarto valor (en general el último) hay que procesarlo fuera del bucle.

```
suma = 0;
cin >> nota;
total_introducidos = 1;

while (nota >= 0 && total_introducidos < 4){
    suma = suma + nota;
    cin >> nota;
    total_introducidos++;
}

suma = suma + nota;
media = suma/total_introducidos;
```



Problema: Si el valor es negativo lo suma. Lo resolvemos con un condicional:

```
suma = 0;
cin >> nota;
total_introducidos = 1;

while (nota >= 0 && total_introducidos < 4){
    suma = suma + nota;
    cin >> nota;
    total_introducidos++;
}

if (nota > 0)
    suma = suma + nota;

media = suma/total_introducidos;
```



Además, no debemos aumentar total_introducidos si es un negativo:

```
if (nota > 0)
    suma = suma + nota;
else
    total_introducidos--;

media = suma/total_introducidos;
```



Podemos observar la complejidad (por no decir chapucería) innecesaria que ha alcanzado el programa.

Replanteamos desde el inicio la solución y usamos variables lógicas:



```
int main(){
    const int TOPE_NOTAS = 4;
    int nota, suma, total_introducidos;
    double media;
    bool tope_alcanzado, es_correcto;

    cout << "Introduzca un máximo de " << TOPE_NOTAS
         << " notas, o cualquier negativo para finalizar.\n ";

    suma = 0;
    total_introducidos = 0;
    es_correcto = true;
    tope_alcanzado = false;

    do{
        cin >> nota;

        if (nota < 0)
            es_correcto = false;
        else{
            suma = suma + nota;
            total_introducidos++;

            if (total_introducidos == TOPE_NOTAS)
                tope_alcanzado = true;
        }
    }while (es_correcto && !tope_alcanzado);

    media = suma/(1.0 * total_introducidos);    // Si total_introducidos es 0
                                                // media = infinito

    if (total_introducidos == 0)
        cout << "\nNo se introdujo ninguna nota";
    else
```

```
    cout << "\nMedia aritmética = " << media;  
}
```

http://decsai.ugr.es/jccubero/FP/II_notas.cpp

Construya los bucles de forma que no haya que arreglar nada después de su finalización

II.2.2.3. Estilo de codificación

La siguiente implementación del anterior algoritmo es nefasta ya que cuesta mucho trabajo entenderla debido a los identificadores elegidos, a las tabulaciones mal hechas y a la falta de líneas en blanco que separen visualmente bloques de código.

```
int main(){
    const int T = 4;
    int v, aux, contador;
    double resultado;
    bool seguir_1, seguir_2;
    aux = 0;
    contador = 0;
    seguir_1 = true;
    seguir_2 = false;
    do{cin >> v;
        if (v < 0)
            seguir_1 = false;
        else{
            aux = aux + v;
            contador++;
            if (contador == T)
                seguir_2 = true;
        }
    }while (seguir_1 && !seguir_2);
    resultado = aux/(1.0*contador);
    if (contador == 0)
        cout << "\nNo se introdujeron valores";
    else
        cout << "\nMedia aritmética = " << resultado;
}
```



II.2.3. Bucles controlador por contador

II.2.3.1. Motivación

Se utilizan para repetir un conjunto de sentencias un número de veces fijado de antemano. Se necesita una variable contadora, un valor inicial, un valor final y un incremento.

Ejemplo. Calcule la media aritmética de cinco enteros leídos desde teclado.

```
int main(){
    int contador, valor, suma, inicio, final;
    double media;

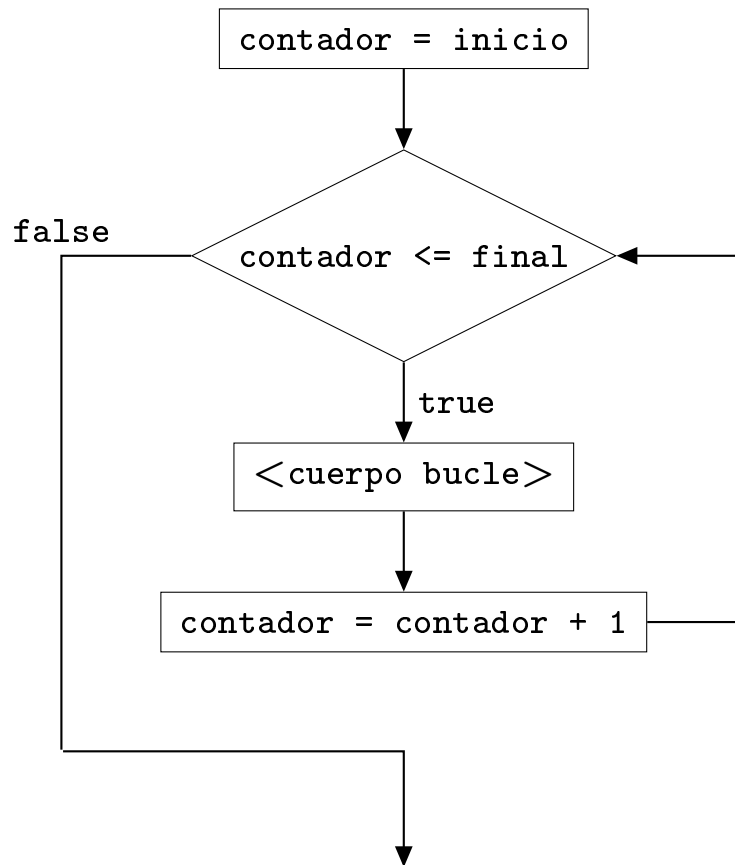
    inicio = 1;
    final = 5;
    suma = 0;
    contador = inicio;

    while (contador <= final){
        cout << "\nIntroduce un número ";
        cin >> valor;
        suma = suma + valor;

        contador = contador + 1;
    }

    media = suma / (final * 1.0);
    cout << "\nLa media es " << media;
}
```

El diagrama de flujo correspondiente es:

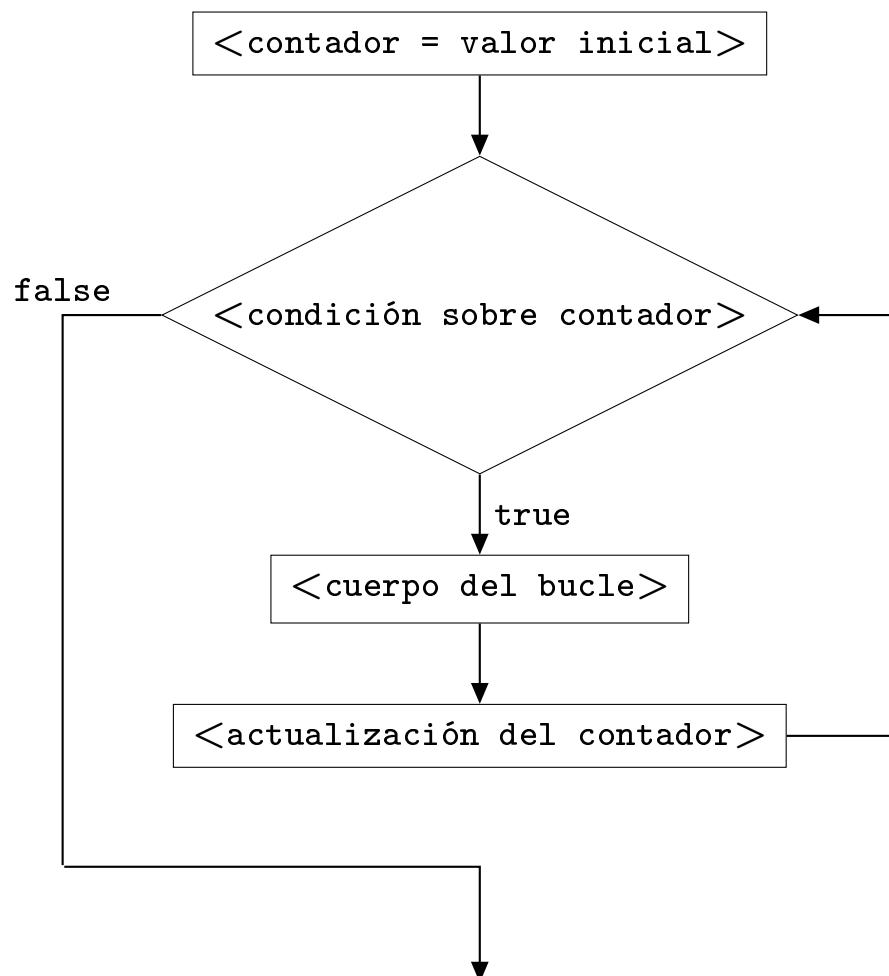


Vamos a implementar el mismo diagrama, pero con otra sintaxis (usando el bucle `for`).

II.2.3.2. Formato

La sentencia `for` permite la construcción de una forma compacta de los ciclos controlados por contador, aumentando la legibilidad del código.

```
for (<contador = valor inicial> ; <condición sobre contador>  
    ; <actualización del contador> )  
    <cuerpo del bucle>
```



Usaremos los bucles `for` cuando sepamos, antes de entrar al bucle, el número de iteraciones que se tienen que ejecutar.

Ejemplo. Calcule la media aritmética de cinco enteros leídos desde teclado.

```
int main(){
    int contador, valor, suma, inicio, final;
    double media;

    inicio = 1;
    final = 5;
    suma = 0;

    for (contador = inicio ; contador <= final ; contador = contador + 1){
        cout << "\nIntroduce un número ";
        cin >> valor;
        suma = suma + valor;
    }

    media = suma / (final*1.0);
    cout << "\nLa media es " << media;
}
```

Como siempre, si sólo hay una sentencia dentro del bucle, no son necesarias las llaves.


```
for (contador = inicio ; contador <= final ; contador = contador + 1){  
    cout << "\nIntroduce un número ";  
    cin >> valor;  
    suma = suma + valor;  
}
```

- ▷ **La primera parte, `contador = inicio`, es la asignación inicial de la variable contadora. Sólo se ejecuta una única vez (cuando entra al bucle por primera vez)**
- ▷ **La segunda parte, `contador <= final`, es la condición de continuación del bucle.**
- ▷ **La tercera parte, `contador = contador + 1`, es la sentencia de actualización de la variable contadora.**

A tener en cuenta:

- ▷ **`contador = contador + 1` aumenta en 1 el valor de `contador` en cada iteración. Por abreviar, suele usarse `contador++` en vez de `contador = contador + 1`**

```
for (contador = inicio ; contador <= final ; contador++)
```

Podemos usar cualquier otro incremento:

```
contador = contador + 4;
```

- ▷ **Si usamos como condición**

```
contador < final
```

habrá menos iteraciones. Si el incremento es 1, se producirá una iteración menos.

- ▷ **También pueden usarse incrementos negativos. En este caso, la condición de terminación del bucle tendrá que ser del tipo `contador >= final` o `contador > final`**

Ejercicio. Queremos imprimir los números del 100 al 1. Encuentre errores en este código:

```
For {x = 100, x>=1, x++}  
    cout << x << " ";
```

Ejemplo. Imprima los pares que hay en el intervalo $[-10, 10]$

```
int candidato;  
  
num_pares = 0;  
  
for (candidato = -10; candidato <= 10; candidato++) {  
    if (candidato % 2 == 0)  
        cout << candidato << " ";  
}
```

Ejercicio. Resuelva el anterior problema con otro bucle distinto.

Ejemplo. Imprima una línea con 10 asteriscos.

```
int i;  
  
for (i = 1; i <= 10; i++)  
    cout << "*";
```

¿Con qué valor sale la variable `i`? 11

Cuando termina un bucle `for`, la variable contadora se queda con el primer valor que hace que la condición del bucle sea falsa.

¿Cuántas iteraciones se producen en un `for`?

▷ Si incremento = 1, $\text{inicio} \leq \text{final}$ y $\text{contador} \leq \text{final}$

`final - inicio + 1`

▷ Si incremento = 1, $\text{inicio} \leq \text{final}$ y $\text{contador} < \text{final}$

`final - inicio`

```
for (i = 0; i < 10; i++)    --> 10 - 0 = 10  
    cout << "*";
```

```
for (i = 12; i > 2; i--)    --> 12 - 2 = 10  
    cout << "*";
```

```
for (i = 10; i >= 1; i--)    --> 10 - 1 + 1 = 10  
    cout << "*";
```

Ampliación:



Número de iteraciones con incrementos cualesquiera.

Si es del tipo `contador <= final`, tenemos que contar cuántos intervalos de longitud igual a `incremento` hay entre los valores `inicio` y `final`. El número de iteraciones será uno más.

En el caso de que `contador < final`, habrá que contar el número de intervalos entre `inicio` y `final - 1`.

El número de intervalos se calcula a través de la división entera.

En resumen, considerando incrementos positivos:

- ▷ Si el bucle es del tipo `contador <= final` el número de iteraciones es $(\text{final} - \text{inicio}) / \text{incremento} + 1$ siempre y cuando sea `inicio <= final`. En otro caso, hay 0 iteraciones.
- ▷ Si el bucle es del tipo `contador < final` el número de iteraciones es $(\text{final} - 1 - \text{inicio}) / \text{incremento} + 1$ siempre y cuando sea `inicio < final`. En otro caso, hay 0 iteraciones.
- ▷ De forma análoga se realizan los cálculos con incrementos negativos (en cuyo caso, el valor inicial ha de ser mayor o igual que el final).

Nota:

Cuando las variables usadas en los bucles no tienen un significado especial podremos usar nombres cortos como `i`, `j`, `k`

Ejercicio. ¿Qué salida producen los siguientes trozos de código?

```
int i, suma_total;  
suma_total = 0;  
  
for (i = 1 ; i <= 10; i++)  
    suma_total = suma_total + 3;
```

```
suma_total = 0;  
  
for (i = 5 ; i <= 36 ; i++)  
    suma_total++;
```

```
suma_total = 0;  
  
for (i = 5 ; i <= 36 ; i = i+1)  
    suma_total++;
```

```
suma_total = 0;  
i = 5;  
  
while (i <= 36){  
    suma_total++;  
    i = i+1;  
}
```

II.2.4. Ámbito de un dato (revisión)

Ya vimos en la página 175 que un dato puede declararse en un bloque condicional y su *ámbito (scope)* es dicho bloque. En general, un dato se puede declarar en cualquier bloque delimitado por { y }. En particular, también en un bloque de una estructura repetitiva:

- ▷ En un bucle controlado por condición, el dato declarado en el bloque no se conoce fuera de él; en particular, no se puede usar en la condición.

```
while (i < 20){           // Error de compilación (i fuera de ámbito)
    int i = 0;
    cout << i << " ";
    i++;
}
```

Además, el dato pierde el valor antiguo en cada iteración:

```
int impresos = 0;

while (impresos < 20){    😞
    int i = 0;
    cout << i << " ";    // Imprime 0 0 0 ...
    impresos++;
    i++;
}
```

Cada vez que entra en el bucle se asigna `i = 0`.

- ▷ El comportamiento es distinto en un bucle `for`. El bucle conserva el valor de la iteración anterior:

```
cout << i;                // Error de compilación (i fuera de ámbito)

for (int i = 0; i < 20; i++)    😊
    cout << i << " "; // Imprime 0 1 2 ...
```

Sólo se asigna `i = 0` en la primera iteración. El uso de este tipo de variables será bastante usual.

Debemos tener cuidado con la declaración de variables con igual nombre que otra definida en un ámbito *superior*.

Prevalece la de ámbito más restringido → *prevalencia de nombre (name hiding)*

```
int main(){
    int suma = 0;                // <- suma (ámbito: main)
    int ultimo;
    int a_sumar = 0;

    cin >> ultimo;

    while (a_sumar < ultimo){
        int suma = 0;            // <- suma (ámbito: bloque while)
        suma = suma + a_sumar;    // <- suma (ámbito: bloque while)
        a_sumar++;
    }

    cout << suma;                // <- suma (ámbito: main)
                                // Imprime 0    😞
}
```

En resumen:

Cuando necesitemos puntualmente variables intermedias para realizar nuestros cálculos, será útil definirlas en el ámbito del bloque de instrucciones correspondiente.

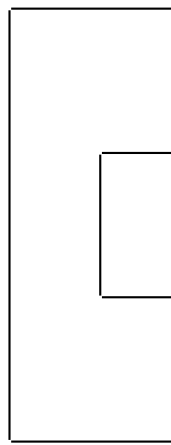
Pero hay que tener cuidado si es un bloque `while` ya que pierde el valor en cada iteración.

Esto no ocurre con las variables contadoras del bucle `for`, que recuerdan el valor que tomó en la iteración anterior.

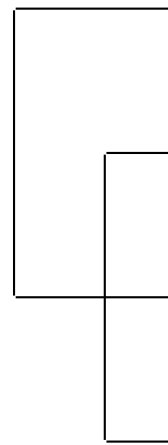
II.2.5. Anidamiento de bucles

Dos bucles se encuentran anidados, cuando uno de ellos está en el bloque de sentencias del otro.

En principio no existe límite de anidamiento, y la única restricción que se debe satisfacer es que deben estar completamente inscritos unos dentro de otros.



(a) Anidamiento correcto



(b) Anidamiento incorrecto

En cualquier caso, un factor determinante a la hora de determinar la rapidez de un algoritmo es la profundidad del anidamiento. Cada vez que anidamos un bucle dentro de otro, la ineficiencia se dispara.

Ejemplo. Imprima la tabla de multiplicar de los `TOPE` primeros números.

```
#include <iostream>
using namespace std;

int main(){
    const int TOPE_IZDA = 3;
    const int TOPE_DCHA = 3;
    int izda, dcha;

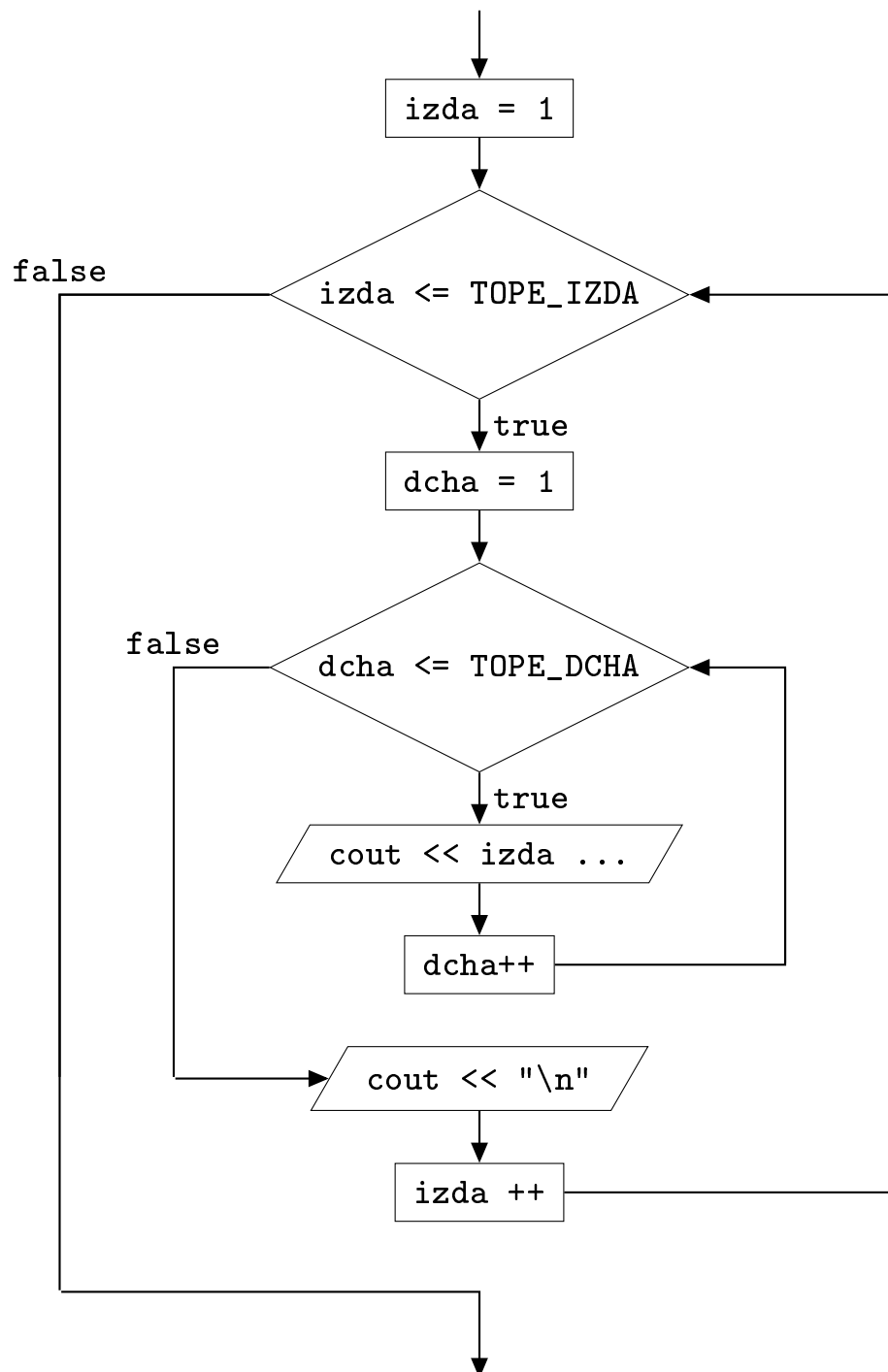
    cout << "Impresión de la tabla de multiplicar "
         << TOPE_IZDA << " x " << TOPE_DCHA << "\n";

    for (izda = 1 ; izda <= TOPE_IZDA ; izda++) {
        for (dcha = 1 ; dcha <= TOPE_DCHA ; dcha++)
            cout << izda << "*" << dcha << "=" << izda * dcha << "  ";
        cout << "\n";
    }
}
```

http://decsai.ugr.es/jccubero/FP/II_tabla_de_multiplicar.cpp

	dcha=1	dcha=2	dcha=3
izda=1	1*1 = 1	1*2 = 2	1*3 = 3
izda=2	2*1 = 2	2*2 = 4	2*3 = 6
izda=3	3*1 = 3	3*2 = 6	3*3 = 9

Observe que cada vez que avanza `izda` y entra de nuevo al bucle, la variable `dcha` vuelve a inicializarse a 1



Al diseñar bucles anidados, hay que analizar cuidadosamente las variables que hay que reiniciar antes de entrar a los bucles más internos

Ejemplo. ¿Qué salida produce el siguiente código?

```
iteraciones = 0;
suma = 0;

for (i = 1 ; i <= n; i++){
    for (j = 1 ; j <= n; j++){
        suma = suma + j;
        iteraciones++;
    }
}
```

Número de iteraciones: n^2

Valor de la variable suma. Supongamos $n = 5$. Valores que va tomando j :

```
1 + 2 + 3 + 4 + 5 +
1 + 2 + 3 + 4 + 5 +
1 + 2 + 3 + 4 + 5 +
1 + 2 + 3 + 4 + 5 +
1 + 2 + 3 + 4 + 5
```

$\text{suma} = 5 * (1 + 2 + 3 + 4 + 5)$. **En general:**

$$\text{suma} = n \sum_{i=1}^{i=n} i = n \frac{n^2 + n}{2} = \frac{n^3 + n^2}{2}$$

Si n es 5, suma se quedará con 75

Ejemplo. ¿Qué salida produce el siguiente código?

```
iteraciones = 0;
suma = 0;

for (i = 1 ; i <= n; i++){
    for (j = i ; j <= n; j++){
        suma = suma + j;
        iteraciones++;
    }
}
```

Número de iteraciones:

$$n + (n - 1) + (n - 2) + \dots + 1 = \sum_{i=1}^{i=n} i = \frac{n^2 + n}{2} < n^2$$

Valor de la variable suma. Supongamos $n = 5$. Valores que va tomando j :

```
1 + 2 + 3 + 4 + 5 +
  2 + 3 + 4 + 5 +
    3 + 4 + 5 +
      4 + 5 +
        5
```

$$\text{suma} = 5 * 5 + 4 * 4 + 3 * 3 + 2 * 2 + 1 * 1 = \sum_{i=1}^{i=n} i^2 = \frac{1}{6}n(n + 1)(2n + 1)$$

Si n es 5, suma se quedará con 55

Ejercicio. Escriba un programa que lea cuatro valores de tipo `char` (`min_izda`, `max_izda`, `min_dcha`, `max_dcha`) e imprima las parejas que pueden formarse con un elemento del conjunto `{min_izda ... max_izda}` y otro elemento del conjunto `{min_dcha ... max_dcha}`. Por ejemplo, si

```
min_izda = b
max_izda = d
```

```
min_dcha = j
max_dcha = m
```

el programa debe imprimir las parejas que pueden formarse con un elemento de `{b c d}` y otro elemento de `{j k l m}`, es decir:

```
bj bk bl bm
cj ck cl cm
dj dk dl dm
```

http://decsai.ugr.es/jccubero/FP/II_Parejas.cpp

Ejemplo. Extienda el ejercicio que comprobaba si un número es primo (página 243) e imprima en pantalla los primos menores que un entero.

```
int main(){
    int entero, posible_primo, divisor, ultimo_divisor_posible;
    bool es_primo;

    cout << "Introduzca un entero ";
    cin >> entero;
    cout << "\nLos primos menores que " << entero << " son:\n";

    /*
    Recorremos todos los números menores que entero
    Comprobamos si dicho número es primo
    */

    for (posible_primo = entero - 1 ; posible_primo > 1 ; posible_primo--){
        es_primo = true;
        divisor = 2;
        ultimo_divisor_posible = posible_primo / 2;

        while (divisor <= ultimo_divisor_posible && es_primo){
            if (posible_primo % divisor == 0)
                es_primo = false;
            else
                divisor++;
        }

        if (es_primo)
            cout << posible_primo << " ";
    }
}
```

http://decsai.ugr.es/jccubero/FP/II_imprimir_primos.cpp

Ejemplo. El *Teorema fundamental de la Aritmética* (Euclides 300 A.C/Gauss 1800) nos dice que podemos expresar cualquier entero como producto de factores primos.

Imprima en pantalla dicha descomposición.

n	primo
360	2
180	2
90	2
45	3
15	3
5	5
1	

Fijamos un valor de primo cualquiera. Por ejemplo `primo = 2`

```
// Dividir n por primo cuantas veces sea posible  
// n es una copia del original
```

```
primo = 2;
```

```
while (n % primo == 0){  
    cout << primo << " ";  
    n = n / primo;  
}
```

Ahora debemos pasar al siguiente primo `primo` **y volver a ejecutar el bloque anterior. Condición de parada:** `n >= primo` **o bien** `n > 1`

```
Mientras n > 1
```

```
    Dividir n por primo cuantas veces sea posible  
    primo = siguiente primo mayor que primo
```


¿Cómo pasamos al siguiente primo?

```
Repite mientras !es_primo
    primo++;
    es_primo = Comprobar si primo es un número primo
```

La comprobación de ser primo o no la haríamos con el algoritmo que vimos en la página 243. Pero no es necesario. Hagamos simplemente `primo++`:

```
/*
Mientras n > 1
    Dividir n por primo cuantas veces sea posible
    primo++
*/

primo = 2;

while (n > 1){
    while (n % primo == 0){
        cout << primo << " ";
        n = n / primo;
    }
    primo++;
}
```

¿Corremos el peligro de intentar dividir `n` por un valor `primo` que no sea primo? No. Por ejemplo, `n=40`. Cuando `primo` sea 4, ¿podrá ser `n` divisible por 4, es decir `n%4==0`? Después de dividir todas las veces posibles por 2, me queda `n=5` que ya no es divisible por 2, ni por tanto, por ningún múltiplo de 2. En general, al evaluar `n%primo`, `n` ya ha sido dividido por todos los múltiplos de `primo`.

Nota. Podemos sustituir `primo++` por `primo = primo+2` (tratando el primer caso `primo = 2` de forma aislada)

```
#include <iostream>
using namespace std;

int main(){
    int entero, n, primo;

    cout << "Descomposición en factores primos";
    cout << "\nIntroduzca un entero ";
    cin >> entero;

    /*
    Copiar entero en n

    Mientras n > 1
        Dividir n por primo cuantas veces sea posible
        primo++
    */

    n = entero;
    primo = 2;

    while (n > 1){
        while (n % primo == 0){
            cout << primo << " ";
            n = n / primo;
        }
        primo++;
    }
}
```

http://decsai.ugr.es/jccubero/FP/II_descomposicion_en_primos.cpp

II.3. Particularidades de C++

C++ es un lenguaje muy versátil. A veces, demasiado ...

II.3.1. Expresiones y sentencias son similares

II.3.1.1. El tipo `bool` como un tipo entero

En C++, el tipo lógico es compatible con un tipo entero. Cualquier expresión entera que devuelva el cero, se interpretará como `false`. Si devuelve cualquier valor distinto de cero, se interpretará como `true`.

```
bool var_logica;

var_logica = false;
var_logica = (4 > 5); // Correcto: resultado false
var_logica = 0;      // Correcto: resultado 0 (false)

var_logica = (4 < 5); // Correcto: resultado true
var_logica = true;
var_logica = 2;      // Correcto: resultado 2 (true)
```

Nota. Normalmente, al ejecutar `cout << false`, se imprime en pantalla un cero, mientras que `cout << true` imprime un uno.

La dualidad entre los tipos enteros y lógicos nos puede dar quebraderos de cabeza en los condicionales

```
int dato = 4;
if (! dato < 5)
    cout << dato << " es mayor o igual que 5";
else
    cout << dato << " es menor de 5";
```



El operador ! tiene más precedencia que <. Por lo tanto, la evaluación es como sigue:

$! \text{ dato} < 5 \Leftrightarrow (!\text{dato}) < 5 \Leftrightarrow (4 \text{ equivale a true}) (!\text{true}) < 5 \Leftrightarrow$
 $\Leftrightarrow \text{false} < 5 \Leftrightarrow 0 < 5 \Leftrightarrow \text{true}$

¡Imprime 4 es mayor o igual que 5!

Para resolver este problema basta usar paréntesis:

```
if (! (dato < 5))
    cout << dato << " es mayor o igual que 5";
else
    cout << dato << " es menor de 5";
```

o mejor, simplificando la expresión siguiendo el consejo de la página 149

```
if (dato >= 5)
    cout << dato << " es mayor o igual que 5";
else
    cout << dato << " es menor de 5";
```

Ejercicio. ¿Qué problema hay en este código? ¿Cómo lo resolvería?

```
bool es_menor;
int min = 3, dato = 2, max = 5;
es_menor = min <= dato <= max;
```

II.3.1.2. El operador de asignación en expresiones

El operador de asignación = se usa en sentencias del tipo:

```
valor = 7;
```

Pero además devuelve un valor: el resultado de la asignación. Así pues, `valor = 7` *es una expresión* que devuelve 7

```
un_valor = otro_valor = valor = 7;
```

Esto producirá fuertes dolores de cabeza cuando por error usemos una expresión de asignación en un condicional:

```
valor = 5;

if (valor = 7)
    <acciones if>    // Siempre se ejecuta este bloque!
else
    <acciones else>

// Además, valor se queda con 7
```



`valor = 7` devuelve 7. Al ser distinto de cero, es `true`. Por tanto, se ejecuta el bloque `if` (y además `valor` se ha modificado con 7)

Otro ejemplo:

```
a = 7;

if (a = 0)
    cout << "\nRaíz= " << -c/b;           // Nunca se ejecuta!
else{
    r1 = -b + sqrt(b*b - 4*a*c) / (2*a) ; // Error lógico
```

II.3.1.3. El operador de igualdad en sentencias

C++ permite que una expresión constituya una sentencia 😞

Esta particularidad no aporta ningún beneficio salvo en casos muy específicos y sin embargo nos puede dar quebraderos de cabeza. Así pues, el siguiente código compila perfectamente:

```
int entero;  
4 + 3;
```

C++ evalúa la expresión **entera** `4 + 3;`, devuelve 7 y no hace nada con él, prosiguiendo la ejecución del programa.

Otro ejemplo:

```
int entero;  
entero == 7;
```

C++ evalúa la expresión **lógica** `entero == 7;`, devuelve `true` y no hace nada con él, prosiguiendo la ejecución del programa.

II.3.1.4. El operador de incremento en expresiones

El operador ++ (y --) puede usarse dentro de una expresión.

Si se usa en forma postfija, primero se evalúa la expresión y luego se incrementa la variable.

Si se usa en forma prefija, primero se incrementa la variable y luego se evalúa la expresión.

Ejemplo. El siguiente condicional expresa una condición del tipo *Comprueba si el siguiente es igual a 10*

```
variable = 9;
if (variable + 1 == 10)
    cout << variable;           // Imprime 9
cout << " " << variable;       // Imprime 9
```

El siguiente condicional expresa una condición del tipo *Comprueba si el actual es igual a 10 y luego increméntalo*

```
variable = 9;
if (variable++ == 10)
    cout << variable;           // No entra
cout << " " << variable;       // Imprime 10
```

El siguiente condicional expresa una condición del tipo *Incrementa el actual y comprueba si es igual a 10*

```
variable = 9;
if (++variable == 10)
    cout << variable;           // Imprime 10
cout << " " << variable;       // Imprime 10
```

Consejo: *Evite el uso de los operadores ++ y -- en la expresión lógica de una sentencia condicional, debido a las sutiles diferencias que hay en su comportamiento, dependiendo de si se usan en formato prefijo o postfijo*



II.3.2. El bucle for en C++

II.3.2.1. Bucles for con cuerpo vacío

El siguiente código no imprime los enteros del 1 al 20. ¿Por qué?

```
for (x = 1; x <= 20; x++);  
    cout << x;
```

Realmente, el código bien tabulado es:

```
for (x = 1; x <= 20; x++)  
    ;  
    cout << x;
```

Evite este tipo de bucles con sentencias vacías ya que son muy oscuros.

II.3.2.2. Bucles for con sentencias de incremento incorrectas

El siguiente código produce un error lógico:

```
for (par = -10; par <= 10; par + 2) // en vez de par = par + 2  
    num_pares++;
```

equivale a:

```
par = -10;  
while (par <= 10){  
    num_pares++;  
    par + 2;  
}
```

Compila correctamente pero la sentencia `par + 2;` no incrementa `par` (recuerde lo visto en la página 278) Resultado: bucle infinito.

II.3.2.3. Modificación del contador

Únicamente mirando la cabecera de un bucle `for` sabemos cuántas iteraciones se van a producir (recuerde lo visto en la página 260). Por eso, en los casos en los que sepamos de antemano cuántas iteraciones necesitamos, usaremos un bucle `for`. En otro caso, usaremos un bucle `while` ó `do while`.

En el caso de que modifiquemos dentro del bucle `for` el valor de la variable controladora o el valor final de la condición, ya no sabremos de antemano cuántas iteraciones se van a ejecutar. Sin embargo, C++ no impone dicha restricción. Será responsabilidad del programador.

Ejercicio. Sume los divisores de `valor`. ¿Dónde está el fallo en el siguiente código? Proponga una solución.

```
suma = 0;
tope = valor/2;

for (divisor = 2; divisor <= tope ; divisor++) {
    if (valor % divisor == 0)
        suma = suma + divisor;

    divisor++;
}
```

Ejemplo. Lea seis notas y calcule la media aritmética. Si se introduce cualquier valor que no esté en el rango $[0, 10]$ se interrumpirá la lectura.

```
double nota, media = 0.0;
int total_introducidos = 0;

for (i = 0; i < 6; i++){
    cin >> nota;

    if (0 <= nota && nota <= 10){
        total_introducidos++;
        media = media + nota;
    }
    else
        i = 7;
}

media = media / total_introducidos;
```



El código produce un resultado correcto de cara al usuario pero es muy oscuro para otro programador. Hay que escudriñar en el cuerpo del bucle para ver todas las condiciones involucradas en la ejecución de éste. Esta información debería estar en la cabecera del `for`.

La solución pasa por usar el bucle `while`:

```
double nota, media = 0.0;
int total_introducidos = 0;
int i;

cin >> nota;
i = 0;

while (i < 6 && 0 <= nota && nota <= 10){
    total_introducidos++;
    media = media + nota;
    i++;
    cin >> nota;
}

media = media / total_introducidos;
```



En resumen:

**No se debe modificar el valor de la variable controladora,
ni el valor final dentro del cuerpo del bucle `for`.**

**En particular, nunca nos saldremos de un bucle `for`
asignándole un valor extremo a la variable contadora.**

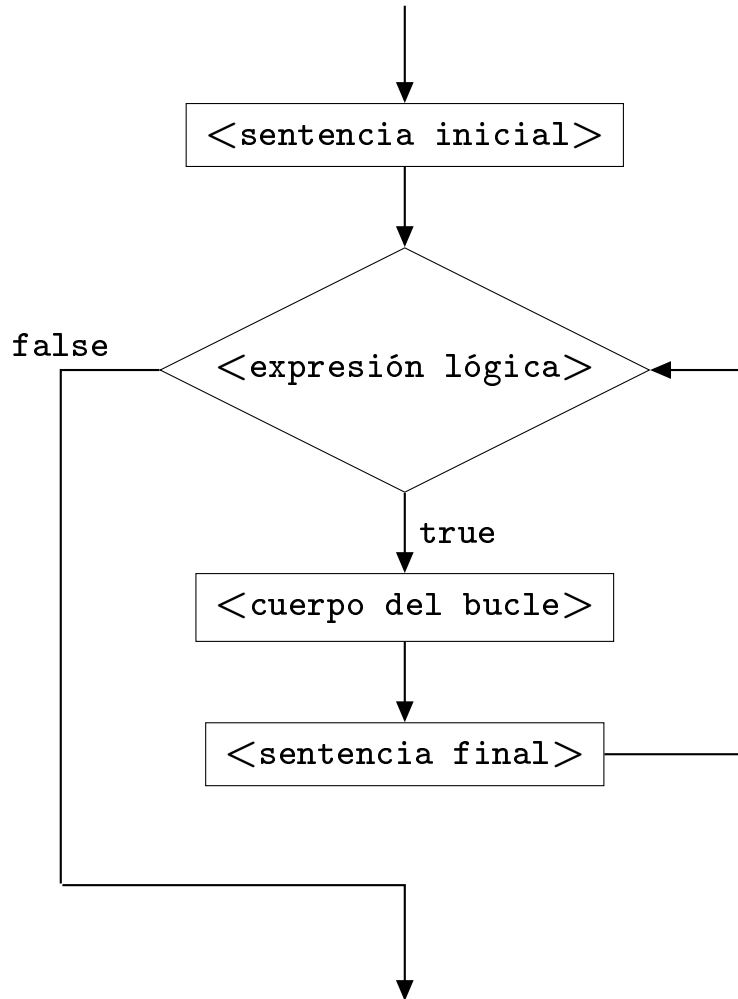


II.3.2.4. El bucle `for` como ciclo controlado por condición

Si bien en muchos lenguajes tales como PASCAL, FORTRAN o BASIC el comportamiento del ciclo `for` es un bucle controlado por contador, en C++ es un ciclo más versátil que permite cualquier tipo de expresiones, involucren o no a un contador:

```
for (<sentencia inicial> ; <expresión lógica>  
    ; <sentencia final> )  
    <cuerpo del bucle>
```

- ▷ **<sentencia inicial>** es la sentencia que se ejecuta antes de entrar al bucle,
- ▷ **<expresión lógica>** es cualquier condición que verifica si el ciclo debe terminar o no,
- ▷ **<sentencia final>** es la sentencia que se ejecuta antes de volver a la expresión lógica para comprobar su valor.



Por tanto, la condición impuesta en el ciclo no tiene por qué ser de la forma `contador < final`, sino que puede ser cualquier tipo de condición. Veamos en qué situaciones es útil esta flexibilidad.

Ejemplo. Retomamos el ejemplo de la media de las notas. Podemos recuperar la versión con un bucle `for` y añadimos a la cabecera la otra condición (la de que la nota esté en el rango correcto). Usamos un `bool`:

```
double nota, media = 0.0;
int total_introducidos = 0;
bool es_nota_correcta = true;

for (i = 0; i < 6 && es_nota_correcta; i++){
    cin >> nota;

    if (0 <= nota && nota <= 10){
        total_introducidos++;
        media = media + nota;
    }
    else
        es_nota_correcta = false;
}

media = media / total_introducidos;
```



Tanto la versión con un bucle `while` de la página 284 como ésta son correctas.

Ejemplo. Compruebe si un número es primo. Lo resolvimos en la página 243:

```
es_primo = true;
divisor = 2;

while (divisor < valor && es_primo){
    if (valor % divisor == 0)
        es_primo = false;
    else
        divisor++;
}
```

Con un for quedaría:

```
es_primo = true;
divisor = 2

for (divisor = 2; divisor < valor && es_primo; divisor++)
    if (valor % divisor == 0)
        es_primo = false;
```

Observe que en la versión con for, la variable `divisor` **siempre** se incrementa, por lo que al terminar el bucle, si el número no es primo, `divisor` será igual al primer divisor de `valor`, más 1.

En los ejemplos anteriores

```
for (i = 0; i < 6 && es_nota_correcta; i++)  
for (divisor = 2; divisor <= tope && es_primo; divisor++)
```

se ha usado dentro del `for` dos condiciones que controlan el bucle:

- ▷ La condición relativa a la variable contadora.
- ▷ Otra condición adicional.

Este código es completamente aceptable en C++.

Consejo: *Limite el uso del bucle `for` en los casos en los que siempre exista:*

- ▷ *Una sentencia de inicialización del contador*
- ▷ *Una condición de continuación que involucre al contador (puede haber otras condiciones **adicionales**)*
- ▷ *Una sentencia final que involucre al contador*



Pero ya puestos, ¿puede usarse entonces, cualquier condición dentro de la cabecera del for? Sí, pero no es recomendable.

Ejemplo. Construya un programa que indique el número de valores que introduce un usuario hasta que se encuentre con un cero (éste no se cuenta)

```
#include <iostream>
using namespace std;

int main(){
    int num_valores, valor;

    cout << "Se contarán el número de valores introducidos";
    cout << "\nIntroduzca 0 para terminar ";

    cin >> valor;
    num_valores = 0;

    while (valor != 0){
        cin >> valor;
        num_valores++;
    }

    cout << "\nEl número de valores introducidos es " << num_valores;
}
```



Lo hacemos ahora con un for:

```
#include <iostream>
using namespace std;

int main(){
    int num_valores, valor;

    cout << "Se contarán el número de valores introducidos";
    cout << "\nIntroduzca 0 para terminar ";

    cin >> valor;

    for (num_valores = 0; valor != 0; num_valores++)
        cin >> valor;

    cout << "\nEl número de valores introducidos es " << num_valores;
}
```



El bucle funciona correctamente pero es un estilo que debemos evitar pues confunde al programador. Si hubiésemos usado otros (menos recomendables) nombres de variables, podríamos tener lo siguiente:

```
for (recorrer = 0; recoger != 0; recorrer++)
```

Y el cerebro de muchos programadores le engañará y le harán creer que está viendo lo siguiente, que es a lo que está acostumbrado:

```
for (recorrer = 0; recorrer != 0; recorrer++)
```

Consejo: Evite la construcción de bucles for en los que la(s) variable(s) que aparece(n) en la condición, no aparece(n) en las otras dos expresiones



Y ya puestos, ¿podemos suprimir algunas expresiones de la cabecera de un bucle `for`? La respuesta es que sí, pero hay que evitarlas SIEMPRE. Oscurecen el código.

Ejemplo. Sume valores leídos desde la entrada por defecto, hasta introducir un cero.

```
int main(){
    int valor, suma;
    cin >> valor;

    for ( ; valor != 0 ; ){
        suma = suma + valor
        cin >> valor;
    }
    .....
}
```



II.3.3. Otras (perniciosas) estructuras de control

Existen otras sentencias en la mayoría de los lenguajes que permiten alterar el flujo normal de un programa.

En concreto, en C++ existen las siguientes sentencias:

`goto` `continue` `break` `exit`



Durante los 60, quedó claro que el uso incontrolado de sentencias de transferencia de control era la principal fuente de problemas para los grupos de desarrollo de software.

Fundamentalmente, el responsable de este problema era la sentencia `goto` que le permite al programador transferir el flujo de control a cualquier punto del programa.

Esta sentencia aumenta considerablemente la complejidad tanto en la legibilidad como en la depuración del código.

Ampliación:

Consulte el libro Code Complete de McConnell, disponible en la biblioteca. En el tema de Estructuras de Control incluye una referencia a un informe en el que se reconoce que un uso inadecuado de un `break`, provocó un apagón telefónico de varias horas en los 90 en NY.



En FP, no se permitirá el uso de ninguna de las sentencias `goto`, `break`, `exit`, `continue`, excepto la sentencia `break` para salir de un `case` dentro de un `switch` (y recuerde lo indicado en la página 174 sobre la fragilidad de la estructura `switch`)



Bibliografía recomendada para este tema:

▷ **A un nivel menor del presentado en las transparencias:**

- Segundo capítulo de Deitel & Deitel
- Capítulos 15 y 16 de McConnell

▷ **A un nivel similar al presentado en las transparencias:**

- Segundo y tercer capítulos de Garrido.
- Capítulos cuarto y quinto de Gaddis.

▷ **A un nivel con más detalles:**

- Capítulos quinto y sexto de Stephen Prata.
- Tercer capítulo de Lafore.

Los autores anteriores presentan primero los bucles junto con la expresiones lógicas y luego los condicionales.

Resúmenes:

Debemos prestar especial atención a los condicionales en los que se asigna un primer valor a alguna variable. Intentaremos garantizar que dicha variable salga siempre del condicional (independientemente de si la condición era verdadera o falsa) con un valor establecido.

Usaremos estructuras condicionales consecutivas cuando los criterios de cada una de ellas sean independientes del resto.

La estructura condicional doble nos permite trabajar con dos condiciones mutuamente excluyentes, comprobando únicamente una de ellas en la parte del `if`. Esto nos permite cumplir el principio de una única vez.



<code>if (cond)</code>		<code>if (cond)</code>
<code>...</code>		<code>...</code>
<code>if (!cond)</code>	→	<code>else</code>
<code>...</code>		<code>...</code>

Utilice las leyes del Álgebra de Boole para simplificar las expresiones lógicas.

La expresión `A || (!A && B)` la sustituiremos por la equivalente a ella: `A || B`

Una situación típica de uso de estructuras condicionales dobles consecutivas, se presenta cuando tenemos que comprobar distintas condiciones independientes entre sí. Dadas n condiciones que dan lugar a n condicionales dobles consecutivos, se pueden presentar 2^n situaciones posibles.



Recordemos que las estructuras condicionales consecutivas se utilizan cuando los criterios de las condiciones son independientes. Por contra, Usaremos los condicionales anidados cuando los criterios sean dependientes entre sí.

<pre>if (c1 && c2 && c3) <accion_A> if (c1 && c2 && !c3) <accion_B> if (c1 && !c2) <accion_C> if (!c1) <accion_D></pre>	→	<pre>if (c1) if (c2) if (c3) <accion_A> else <accion_B> else <accion_C> else <accion_D></pre>
		

Usaremos los condicionales dobles y anidados de forma coherente para no duplicar ni el código de las sentencias ni el de las expresiones lógicas de los condicionales, cumpliendo así el principio de una única vez.

Si debemos comprobar varias condiciones, todas ellas mutuamente excluyentes entre sí, usaremos estructuras condicionales dobles anidadas

Cuando `c1`, `c2` y `c3` sean mutuamente excluyentes:

<code>if (c1)</code>		<code>if (c1)</code>
<code>...</code>		<code>...</code>
<code>if (c2)</code>		<code>else if (c2)</code>
<code>...</code>	→	<code>...</code>
<code>if (c3)</code>		<code>else if (c3)</code>
<code>...</code>		<code>...</code>
		

El compilador no comprueba que cada `case` termina en un `break`. Es por ello, que debemos evitar, en la medida de lo posible, la sentencia `switch` y usar condicionales anidados en su lugar.

El uso de variables intermedias para determinar criterios nos ayuda a no repetir código y facilita la legibilidad de éste. Se les asigna un valor en un bloque y se observa su contenido en otro.

El uso de variables intermedias nos ayuda a separar los bloques de E/S y C. Se les asigna un valor en un bloque y se observa su contenido en otro.

El tipo enumerado puede verse como una extensión del tipo `bool` ya que nos permite manejar más de dos opciones excluyentes.

Al igual que un `bool`, un dato enumerado contendrá un único valor en cada momento. La diferencia está en que con un `bool` sólo tenemos 2 posibilidades y con un enumerado tenemos más (pero no tantas como las 256 de un `char`, por ejemplo).

En el diseño de los bucles siempre hay que comprobar el correcto funcionamiento en los casos extremos (primera y última iteración)

Para no repetir código en los bucles que leen datos, normalmente usaremos la técnica de lectura anticipada:

```
cin >> dato;

while (dato no es último){
    procesar_dato
    cin >> dato;
}
```

A veces también puede ser útil introducir variables lógicas que controlen la condición de terminación de lectura:

```
do{
    cin >> dato;

    test_dato = ¿es el último?

    if (test_dato)
        procesar_dato
}while (test_dato);
```

Evite, en la medida de lo posible, preguntar por algo que sabemos de antemano que sólo va a ser verdadero en la primera iteración.

Debemos garantizar que en algún momento, la condición del bucle se hace falsa.

Tenga especial cuidado dentro del bucle con los condicionales que modifican alguna variable presente en la condición.

En la construcción de bucles con condiciones compuestas, empiece planteando las condiciones simples que la forman. Piense cuándo queremos salir del bucle y conecte adecuadamente dichas condiciones simples, dependiendo de si nos queremos salir cuando todas simultáneamente sean `false` (conectamos con `||`) o cuando cualquiera de ellas sea `false` (conectamos con `&&`)

Diseñe las condiciones compuestas de forma que sean fáciles de leer: en muchas situaciones, nos ayudará introducir variables lógicas a las que se les asignará un valor para salir del bucle cuando se verifique cierta condición de parada.

Si una vez que termina un bucle controlado por varias condiciones, necesitamos saber cuál de ellas hizo que terminase éste, introduciremos variables intermedias para determinarlo. Nunca repetiremos la evaluación de condiciones.

Construya los bucles de forma que no haya que arreglar nada después de su finalización

Usaremos los bucles `for` cuando sepamos, antes de entrar al bucle, el número de iteraciones que se tienen que ejecutar.

Cuando necesitemos puntualmente variables intermedias para realizar nuestros cálculos, será útil definirlas en el ámbito del bloque de instrucciones correspondiente.

Pero hay que tener cuidado si es un bloque `while` ya que pierde el valor en cada iteración.

Esto no ocurre con las variables contadoras del bucle `for`, que recuerdan el valor que tomó en la iteración anterior.

Al diseñar bucles anidados, hay que analizar cuidadosamente las variables que hay que reiniciar antes de entrar a los bucles más internos

Consejo: *En aquellos casos en los que debe asignarle un valor a una variable `bool` dependiendo del resultado de la evaluación de una expresión lógica, utilice directamente la asignación de dicha expresión en vez de un condicional doble.*



Consejo: *En los condicionales que simplemente comprueban si una variable lógica contiene `true`, utilice el formato `if (variable_logica)`. Si se quiere consultar si la variable contiene `false` basta poner `if (!variable_logica)`*



Consejo: *Fomente el uso de los bucles pre-test. Lo usual es que haya algún caso en el que no queramos ejecutar el cuerpo del bucle ni siquiera una vez.*



Consejo: *Evite el uso de los operadores `++` y `--` en la expresión lógica de una sentencia condicional, debido a las sutiles diferencias que hay en su comportamiento, dependiendo de si se usan en formato prefijo o postfijo*



Consejo: *Limite el uso del bucle `for` en los casos en los que siempre exista:*



- ▷ *Una sentencia de inicialización del contador*
- ▷ *Una condición de continuación que involucre al contador (puede haber otras condiciones **adicionales**)*
- ▷ *Una sentencia final que involucre al contador*

Consejo: *Evite la construcción de bucles `for` en los que la(s) variable(s) que aparece(n) en la condición, no aparece(n) en las otras dos expresiones*



Consejo: *Procurad no abusar de este estilo de codificación, es decir, evitad la construcción de bucles `for` sin ninguna sentencia en su interior*



Destaque visualmente el bloque de instrucciones de una estructura condicional.



Diseñe los algoritmos para que sean fácilmente extensibles a situaciones más generales.



Analice siempre con cuidado las tareas a resolver en un problema e intente separar los bloques de código que resuelven cada una de ellas.



Los bloques de código que realizan entradas o salidas de datos (`cin`, `cout`) estarán separados de los bloques que realizan cálculos.



Los algoritmos que realizan una búsqueda, deben salir de ésta en cuanto se haya encontrado el valor. Normalmente, usaremos una variable lógica para controlarlo.

IMPORTANT

Use descripciones de algoritmos que sean CONCISAS con una presentación visual agradable y esquemática.

**No se hará ningún comentario de aquello que sea obvio.
Más vale poco y bueno que mucho y malo.**

Las descripciones serán de un BLOQUE completo.

Sólo se usarán comentarios al final de una línea en casos puntuales, para aclarar el código de dicha línea.



Jamás usaremos un tipo string para detectar un número limitados de alternativas posibles ya que es propenso a errores.



No se debe modificar el valor de la variable controladora, ni el valor final dentro del cuerpo del bucle `for`.

En particular, nunca nos saldremos de un bucle `for` asignándole un valor extremo a la variable contadora.



En FP, no se permitirá el uso de ninguna de las sentencias goto, break, exit, continue, excepto la sentencia break para salir de un case dentro de un switch (y recuerde lo indicado en la página 174 sobre la fragilidad de la estructura switch)



Principio de Programación:

Sencillez (Simplicity)

Fomente siempre la sencillez y la legibilidad en la escritura de código



Tema III

Vectores y Matrices

Objetivos:

- ▷ Introduciremos los vectores de C++ que nos permitirán almacenar conjuntamente datos del mismo tipo.
- ▷ Veremos varios algoritmos de búsqueda y ordenación.
- ▷ Introduciremos las matrices (vectores de varias dimensiones).

III.1. Fundamentos

III.1.1. Introducción

III.1.1.1. Motivación

Ejemplo. Lea tres notas desde teclado y diga cuántos alumnos superan la media

```
#include <iostream>
using namespace std;

int main(){
    int superan_media;
    double nota1, nota2, nota3, media;

    cout << "Introduce nota 1: ";
    cin >> nota1;
    cout << "Introduce nota 2: ";
    cin >> nota2;
    cout << "Introduce nota 3: ";
    cin >> nota3;

    media = (nota1 + nota2 + nota3) / 3.0;
    superan_media = 0;

    if (nota1 > media)
        superan_media++;

    if (nota2 > media)
        superan_media++;
```



```

    if (nota3 > media)
        superan_media++;

    cout << superan_media << " alumnos han superado la media\n";
}

```

Problema: ¿Qué sucede si queremos almacenar las notas de 50 alumnos? Número de variables imposible de sostener y recordar.

Solución: Introducir un tipo de dato nuevo que permita representar dichas variables en una única *estructura de datos*, reconocible bajo un nombre único.

Un **vector (array)** es un tipo de dato, compuesto de un número fijo de componentes del mismo tipo y donde cada una de ellas es directamente accesible mediante un índice. El índice será un entero, siendo el primero el 0.

	notas[0]	notas[1]	notas[2]
notas =	2.4	4.9	6.7

III.1.1.2. Declaración

```
<tipo> <identificador> [<núm. componentes>];
```

- ▷ **<tipo>** indica el tipo de dato común a todas las **componentes (elements/components)** del vector.
- ▷ **<núm. componentes>** determina el número de componentes del vector, al que llamaremos **tamaño (size)** del vector. El número de componentes debe conocerse en el momento de la declaración y no es posible alterarlo durante la ejecución del programa. Pueden usarse literales ó constantes enteras (char, int, ...), pero nunca una variable (en C sí, pero no en C++)

En C++ no se puede declarar un vector con un tamaño variable (En C99 sí se puede)

Nota:

Algunos compiladores permiten el uso de variables para dimensionar un vector, pero el estándar de C++ lo prohíbe.

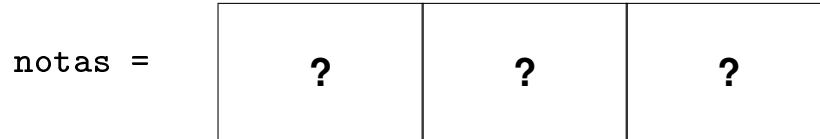
```
int main(){  
    const int MAX_ALUMNOS = 3;  
    double notas[MAX_ALUMNOS];  
    int variable = 3;  
    double notas[variable]; // Error de compilación
```

Consejo: *Fomente el uso de constantes en vez de literales para especificar el tamaño de los vectores.*



- ▷ Las componentes ocupan posiciones contiguas en memoria.

```
const int MAX_ALUMNOS = 3;
double notas[MAX_ALUMNOS];
```



Otros ejemplos:

```
int main(){
    const int NUM_REACTORES = 20;

    int    TemperaturasReactores[NUM_REACTORES]; // Correcto.
    bool   casados[40];                          // Correcto.
    char   NIF[8];                                // Correcto.
    .....
}
```

III.1.2. Operaciones básicas

III.1.2.1. Acceso

Dada la declaración:

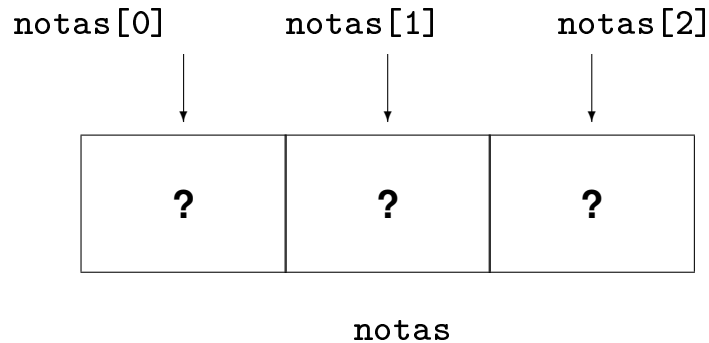
<tipo> <identificador> [<núm. componentes>];

A cada componente se accede de la forma:

<identificador> [<índice>]

- ▷ El índice de la primera componente del vector es 0.
El índice de la última componente es <núm. componentes> - 1

```
const int TOTAL_ALUMNOS = 3;  
double notas[TOTAL_ALUMNOS];
```



`notas[9]` y `notas['3']` no son componentes correctas.

- ▷ Podemos acceder a cualquier componente en cualquier momento: diremos que es un **acceso directo (direct access)**. En Programación y en otras disciplinas de Informática, este tipo de acceso se conoce también como **acceso aleatorio (random access)**.
- ▷ Cada componente **es una variable** más del programa, del tipo indicado en la declaración del vector.

Por ejemplo, `notas[0]` es una variable de tipo `double`.

Por lo tanto, con dichas componentes podremos realizar todas las operaciones disponibles (asignación, lectura con `cin`, pasarlas como parámetros actuales, etc). Lo detallamos en los siguientes apartados.

Las componentes de un vector son datos como los vistos hasta ahora, por lo que le serán aplicables las mismas operaciones definidas por el tipo de dato asociado.

III.1.2.2. Asignación

- ▷ **No se permiten asignaciones globales sobre todos los elementos del vector (salvo en la inicialización, como veremos posteriormente). Las asignaciones se deben realizar componente a componente.**

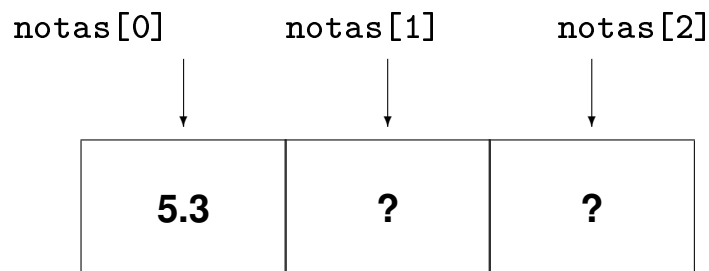
```
int main(){
    const int MAX_ALUMNOS = 3;
    double notas[MAX_ALUMNOS];

    notas = {1,2,3};    // Error de compilación
```

- ▷ **Asignación componente a componente:**

<identificador> [<índice>] = <expresión>;

<expresión> **ha de ser del mismo tipo que el definido en la declaración del vector (o al menos *compatible*).**



```
int  main(){
    const int MAX_ALUMNOS = 3;
    double notas[MAX_ALUMNOS];
    double una_nota;

    notas[0] = 5.3;           // Correcto. double = double
    notas[1] = 7;             // Correcto. double = int
    una_nota = notas[1];      // Correcto. double = double
```

- ▷ **Cuando una componente aparece a la izquierda y derecha de un**

asignación, para simplificar la escritura de expresiones, podemos hacer uso de lo visto en la página 50:

```
notas[0] = notas[0] + 0.5;
```

Equivale a:

```
notas[0] += 0.5;
```

▷ **El índice puede ser una variable entera.**

```
int main(){
    const int MAX_ALUMNOS = 3;
    double notas[MAX_ALUMNOS];

    notas[0] = 5.3;

    int i = 0;
    notas[i] = 5.3;
```

Esto nos permitirá recorrer el vector con un bucle. Lo vemos en el siguiente apartado.

III.1.2.3. Lectura y escritura

La lectura y escritura se realiza componente a componente. Para leer con `cin` o escribir con `cout` *todas* las componentes utilizaremos un bucle:

```
for (int i = 0; i < MAX_ALUMNOS; i++){
    cout << "Introducir nota del alumno " << i << ": ";
    cin >> notas[i];
}

media = 0;

for (int i = 0; i < MAX_ALUMNOS; i++){
    media = media + notas[i];
}

media = media / MAX_ALUMNOS;

cout << "\nMedia = " << media;
}
```

III.1.2.4. Inicialización

C++ permite inicializar una variable vector en la declaración.

▷ `int vector[3];`

Los tres datos tienen un valor indeterminado

`vector[0]=?, vector[1]=?, vector[2]=?`

▷ `int vector[3] = {4,5,6};`

inicializa `vector[0]=4, vector[1]=5, vector[2]=6`

▷ `int vector[7] = {8};`

***¡Cuidado!* Declara un vector de 7 componentes e inicializa la primera a 8 (sólo la primera) y el resto a cero. Uno esperaría que esta sentencia inicializase todas las componentes a 8, pero no es así. Por lo tanto, en la medida de lo posible, evite este tipo de inicializaciones y use un bucle para la asignación de los valores.**

▷ `int vector[7] = {3,5};`

inicializa `vector[0]=3, vector[1]=5` y el resto se inicializan a cero.

▷ `int vector[7] = {0};`

inicializa todas las componentes a cero.

▷ `int vector[] = {1,3,9};`

automáticamente el compilador asume `int vector[3]`

Si queremos declarar un vector de componentes constantes, pondremos el cualificador `const` en la definición del vector y a continuación habrá que, obligatoriamente, inicializarlo.

`const int vector[3] = {4,5,6};`

III.1.3. Trabajando con las componentes

III.1.3.1. Representación en memoria

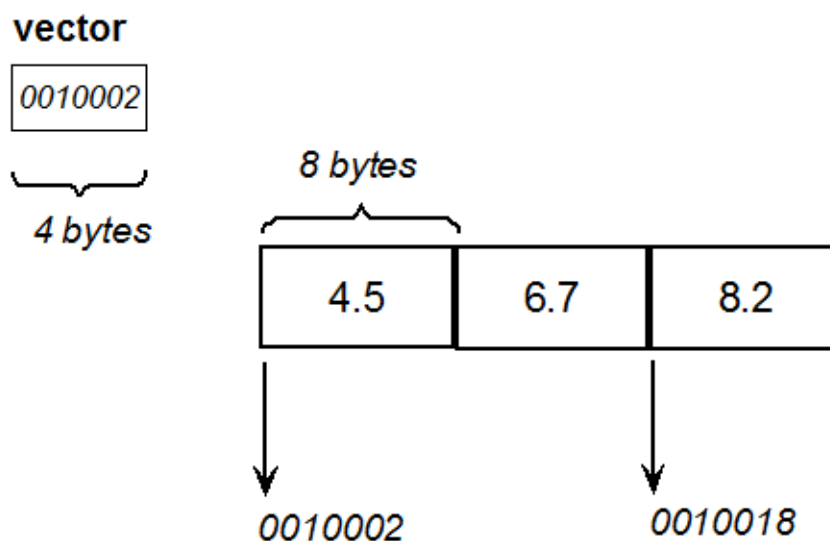
¿Qué ocurre si accedemos a una componente fuera de rango?

```
cout << notas[15];    // Posible error grave en ejecución!  
notas[15] = 5.3;      // Posible error grave en ejecución!
```



El compilador trata a un vector como un dato constante que almacena la dirección de memoria donde empiezan a almacenarse las componentes. En un SO de 32 bits, para guardar una dirección de memoria, se usan 32 bits (4 bytes).

```
int main(){  
    double vector[3];  
  
    vector[0] = 4.5;  
    vector[1] = 6.7;  
    vector[2] = 8.2;
```



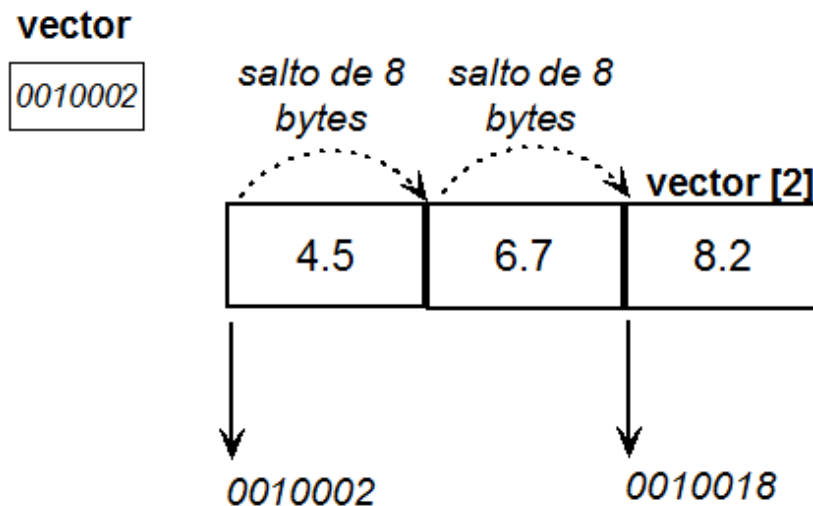
Es como si el programador hiciese la asignación siguiente:

```
vector = 0010002;
```

Realmente, dicha asignación es realizada por el compilador que considera `vector` como un dato constante:

```
double vector[3];    // Compilador -> vector = 0010002;  
  
vector = 00100004;   // Error compilación. vector es cte.  
vector[0] = 4.5;     // Correcto
```

Para saber dónde está la variable `vector[2]`, el compilador debe dar dos *saltos*, a partir de la dirección de memoria `0010002`. Cada salto es de 8 bytes (suponiendo que los `double` ocupan 8 bytes). Por tanto, se va a la variable que empieza en la dirección `0010018`



En general, si declaramos

```
TipoBase vector[TAMANIO];
```

para poder acceder a `vector[indice]`, el compilador dará un total de `indice` saltos tan grandes como diga el `TipoBase`, a partir de la primera posición de `vector`, es decir:

$$\text{vector[indice]} \equiv \text{vector} + \text{indice} * \text{sizeof}(\text{<tipo base>})$$

```
int main(){  
    const int MAX_ALUMNOS = 3;  
    double vector[MAX_ALUMNOS];  
  
    vector[15] = 5.3;        // Posible error grave en ejecución!
```



Para aumentar la eficiencia, el compilador no comprueba que el índice de acceso a las componentes esté en el rango correcto, por lo que cualquier acceso de una componente con un índice fuera de rango tiene consecuencias imprevisibles.

Nota:

El simple acceso a la componente para observar su valor, aún sin modificarla, puede ocasionar un error durante la ejecución. El estándar de C++ indica que un [acceso fuera de rango \(out of bound access\)](#) a una componente de un vector provocan un [comportamiento indeterminado \(undefined behaviour\)](#) .

¿Qué significa comportamiento indeterminado? Que el creador del compilador es libre de decidir la respuesta a dar ante esa situación. Puede decidir, por ejemplo, ¡formatear el disco duro!

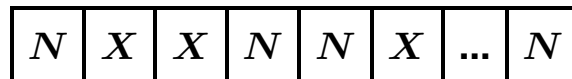
III.1.3.2. Gestión de componentes utilizadas

Dado un vector, ¿cómo gestionamos el uso de sólo una parte del mismo?
¿Dónde debemos colocar los *huecos*?

Supongamos que podemos elegir un valor especial *nulo* (N) del tipo de dato de las componentes para indicar que la componente no está siendo usada. Por ejemplo, en un vector de tipo de dato `string`, el valor nulo N podría ser la cadena "", o -1 en un vector de positivos.

A partir de ahora X denotará una componente utilizada con un valor concreto. Tendríamos las siguientes opciones:

► *Huecos en cualquier sitio y marcados con un valor especial N*



El recorrido será del tipo:

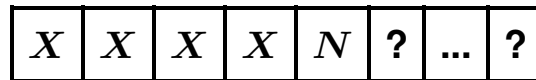
```
for (int i = 0; i < MAX; i++){  
    if (vector[i] !=  $N$ ){  
        .....  
    }  
}
```

En el ejemplo de las notas:

```
const double NULO = -1.0;  
  
for (int i = 0; i < MAX_ALUMNOS; i++){  
    if (notas[i] != NULO)  
        suma = suma + notas[i];  
}
```

► **Todos los huecos a la derecha. El primero, marcado con un valor especial N**

Dejamos todos los huecos juntos (normalmente en la zona de índices altos, es decir, a la derecha) Ponemos N en la última componente utilizada (debemos reservar siempre una componente para N).



El recorrido será del tipo:

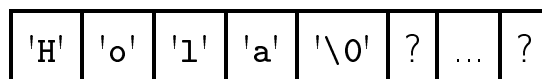
```
for (int i=0; vector[i] !=  $N$  ; i++){  
    .....  
}
```

En el ejemplo de las notas:

```
const double NULO = -1.0;  
i = 0;  
  
while (notas[i] != NULO){  
    suma = suma + notas[i];  
    i++;  
}
```

Ampliación:

Con vectores de tipo `char` es habitual usar como valor nulo el literal `'\0'`. Los vectores de `char` manipulados de esta forma se denominan [cadenas de C](#) ([C string](#)) .



Pero no siempre podemos elegir un valor especial del tipo de dato usado en el vector para señalar un hueco. En este caso, tenemos varias posibilidades:

► *Todos los huecos a la derecha, sin usar un valor especial*

En este caso, basta usar una variable entera, `util` que indique el número de componentes usadas.

`util_vector = 4`

X	X	X	X	?	?	...	?
---	---	---	---	---	---	-----	---

Los índices de las componentes utilizadas irán desde 0 hasta `util-1`:

```
for (int i=0; i < util_vector ; i++){  
    .....  
}
```

En el ejemplo de las notas:

```
util_notas = 3;  
  
for (int i = 0; i < util_notas; i++)  
    suma = suma + notas[i];
```

Ésta será una de las formas más habituales con las que trabajaremos con un vector.

► Huecos en cualquier sitio

?	X	X	?	?	X	...	?
---	---	---	---	---	---	-----	---

¿Cómo recorreremos el vector? En el ejemplo de las notas:

?	9.5	4.6	?	?	7.8	...	?
---	-----	-----	---	---	-----	-----	---

¿Sería válido el siguiente bucle?

```
for (int i = 0; i < MAX_ALUMNOS; i++)  
    suma = suma + notas[i];
```

Este bucle no nos vale ya que también procesa las componentes no asignadas (con un valor indeterminado)

Podemos usar otro vector de `bool` que nos indique si la componente está asignada o no:

```
bool utilizado_en_notas[MAX_ALUMNOS];
```

false	true	true	false	false	true	...	false
-------	------	------	-------	-------	------	-----	-------

```
for (int i = 0; i < MAX_ALUMNOS; i++){  
    if (utilizado_en_notas[i])  
        suma = suma + notas[i];  
}
```

Este bucle sí sería correcto, aunque nos obliga a trabajar con dos vectores en paralelo. Por lo tanto, siempre que podamos, elegiremos la primera opción (todos los huecos a la derecha), para evitar trabajar con dos vectores dependientes entre sí (el vector que contiene los datos y el que indica los elementos utilizados).

En resumen, los tipos más comunes de gestión de un vector son:

- ▷ Si podemos elegir un valor especial N que represente componente no utilizada.

- Dejando huecos entre las componentes:

N	X	X	N	N	X	...	N
-----	-----	-----	-----	-----	-----	-----	-----

Los recorridos deben comprobar si la componente actual es N

- Sin dejar huecos (se colocan todas al principio, por ejemplo)
Ponemos N en la última componente utilizada (debemos reservar siempre una componente para N).

X	X	X	X	N	?	...	?
-----	-----	-----	-----	-----	---	-----	---

- ▷ Si no podemos elegir dicho valor N

- Dejando huecos entre las componentes. Necesitaremos identificar las componentes no utilizadas usando, por ejemplo, un vector de `bool`.

?	X	X	?	?	X	...	?
---	-----	-----	---	---	-----	-----	---

false	true	true	false	false	true	...	false
-------	------	------	-------	-------	------	-----	-------

Los recorridos deben comprobar si la componente actual está utilizada viendo el valor correspondiente en el vector de `bool`

- Sin dejar huecos (se colocan todas al principio, por ejemplo)
Usamos una variable entera que nos indique el número de componentes usadas.

`util = 4`

X	X	X	X	?	?	...	?
-----	-----	-----	-----	---	---	-----	---

Nota:

Una forma mixta de procesar los datos es trabajar con una representación con huecos y cada cierto tiempo, compactarlos para eliminar dichos huecos.

Ejemplo. Retomamos el ejemplo de las notas y leemos desde teclado el número de alumnos que vamos a procesar. Calculamos el número de alumnos que superan la media aritmética de las notas.

Optamos por una representación en la que todas las componentes usadas estén correlativas. Alternativas:

- ▷ Usar como valor nulo el -1, ya que no es una nota posible.

8.2	9.1	3.5	-1	?	?	...	?
-----	-----	-----	----	---	---	-----	---

- ▷ Usar una variable adicional que indique el número de alumnos actuales.

util_notas = 3							
8.2	9.1	3.5	?	?	?	...	?

Optamos por esta última.

```
int main(){
    const int MAX_ALUMNOS = 100;
    double notas[MAX_ALUMNOS];
    int util_notas, superan_media;
    double media;

    cout << "Introduzca el número de alumnos (entre 1 y " +
            to_string(MAX_ALUMNOS) + "): ";

    do{
        cin >> util_notas;
    }while (util_notas <= 0 || util_notas > MAX_ALUMNOS);

    for (int i=0; i < util_notas; i++){
        cout << "nota[" << i << "] --> ";
        cin >> notas[i];
    }
```

```
}  
media = 0;  
  
for (int i = 0; i < util_notas; i++)  
    media = media + notas[i];  
  
media = media / util_notas;  
superan_media = 0;  
  
for (int i = 0; i < util_notas; i++){  
    if (notas[i] > media)  
        superan_media++;  
}  
  
cout << "\n" << superan_media << " alumnos han superado la media\n";  
}
```

http://decsai.ugr.es/jccubero/FP/III_notas.cpp

Ejemplo. Representemos la ocupación de un autobús con un vector de cadenas de caracteres.

- ▷ Los asientos se numeran a partir de 1. El asiento 0 corresponde al conductor.
- ▷ Se permite que haya asientos no ocupados entre los pasajeros y permitimos acceder a cualquier componente (asiento) en cualquier momento. Para identificar un asiento vacío usamos $N = ""$

"Pedro"	"Juan"	""	"Carlos"	""	""	...	"María"
---------	--------	----	----------	----	----	-----	---------

X	X	N	X	N	N	...	X
---	---	---	---	---	---	-----	---

```
int main(){
    const string TERMINADOR = "-";
    const string VACIO = "";
    const int MAX_PLAZAS = 50;
    string pasajeros[MAX_PLAZAS];      // C++ inicializa los string a ""
    string conductor, nombre_pasajero;
    int    asiento;

    for (int i=0; i < MAX_PLAZAS; i++)
        pasajeros[i] = VACIO;

    cout << "Autobús.\n";
    cout << "\nIntroduzca nombre del conductor: ";
    cin >> conductor;

    pasajeros[0] = conductor;

    cout << "\nIntroduzca los nombres de los pasajeros y su asiento."
         << "Termine con " << TERMINADOR << "\n";
    cout << "\nNombre: ";
```

```
cin >> nombre_pasajero;

while (nombre_pasajero != TERMINADOR){
    cout << "Asiento: ";
    cin >> asiento;

    pasajeros[asiento] = nombre_pasajero;

    cout << "\nNombre: ";
    cin >> nombre_pasajero;
}

cout << "\n\nConductor: " << pasajeros[0];

for (int i=1; i < MAX_PLAZAS; i++){
    cout << "\nAsiento número: " << i;

    if (pasajeros[i] != VACIO)
        cout << " Pasajero: " << pasajeros[i];
    else
        cout << " Vacío";
}
}
```

Nota:

Al ejecutar el programa, deben introducirse cadenas de caracteres sin espacios en blanco. En el siguiente tema veremos cómo leer cadenas sin esta restricción.

http://decsai.ugr.es/jccubero/FP/III_autobus.cpp

III.1.3.3. El tipo string como secuencia de caracteres

El tipo de dato `string` es algo más complejo de lo que se ha visto hasta ahora (se verá que, realmente, es una clase con métodos asociados)

Una particularidad de este tipo es que podemos acceder a sus componentes con la notación corchete de los vectores. Cada componente es de tipo `char` y podemos acceder a ella para observar o modificar su valor: al igual que ocurría con los vectores, si el índice no es correcto, se producirá un comportamiento indeterminado (página 318). Así pues, no debemos modificar a través del acceso con corchetes componentes que no estuviese. ya incluidas en el `string`.

```
#include <iostream>
#include <string>
using namespace std;

int main(){
    string cadena;

    cadena = "FundamentAs de Programación";
    cout << cadena[0]; // Imprime F
    cadena[9] = 'o';   // Fundamentos de Programación
    cadena[30] = 'o';  // Comportamiento indeterminado
}
```

Aunque no entendamos por ahora la sintaxis, introducimos algunas operaciones útiles con datos de tipo `string`. Supongamos la siguiente declaración:

```
string cadena;  
char character;
```

▷ `cadena.size()` devuelve el tamaño actual de la cadena.

Por defecto, C++ inicializa un dato de tipo `string` a la **cadena vacía** (*empty string*) que es el literal `""`. El tamaño es cero.

▷ `cadena.push_back(character)` añade un carácter al final de la cadena.

▷ `+` es un operador binario que concatena dos cadenas.

```
#include <iostream>  
#include <string>  
using namespace std;
```

```
int main(){  
    string cadena;  
  
    cadena = "Fundamento";  
    cout << cadena.size();    // 10  
    cadena.push_back('s');  
    cout << cadena;           // Fundamentos  
    cout << cadena.size();    // 11  
    cadena = cadena + " de Programación";  
    cout << cadena;           // Fundamentos de Programación  
    cadena[50] = 'o';         // Comportamiento indeterminado
```



No *añada* caracteres a un `string` mediante el acceso directo a la componente con la notación corchete. Use `push_back` en su lugar.

III.2. Recorridos sobre vectores

En lo que sigue, asumiremos un vector sin huecos y sin destacar ninguna componente como valor nulo.

util = 4

X	X	X	X	?	?	...	?
---	---	---	---	---	---	-----	---

Sin pérdida de generalidad, trabajaremos sobre un vector de `char`.

III.2.1. Algoritmos de búsqueda

Los algoritmos de búsqueda tienen como finalidad localizar la posición de un elemento específico en un vector.

- ▷ **Búsqueda secuencial (Linear/Sequential search)** . Técnica más sencilla.
- ▷ **Búsqueda binaria (Binary search)** . Técnica más eficiente, aunque requiere que el vector esté ordenado.

En los siguientes algoritmos usaremos una variable `pos_encontrado`. Si el valor se encuentra, le asignaremos la posición del vector en la que ha sido encontrado. En caso contrario, le asignaremos un valor imposible de posición, como por ejemplo -1.

III.2.1.1. Búsqueda Secuencial

Algoritmo: Primera Ocurrencia

Ir recorriendo las componentes del vector

- mientras no se encuentre el elemento buscado Y
- mientras no lleguemos al final del mismo

```
const int TAMANIO = 50;
char vector[TAMANIO];
.....
i = 0;
pos_encontrado = -1;
encontrado = false;

while (i < total_utilizados && !encontrado){
    if (vector[i] == buscado){
        encontrado = true;
        pos_encontrado = i;
    }
    else
        i++;
}

if (encontrado)
    cout << "\nEncontrado en la posición " << pos_encontrado;
else
    cout << "\nNo encontrado";
```

http://decsai.ugr.es/jccubero/FP/III_vector_busqueda_secuencial.cpp



Verificación (pruebas de unidad):

- ▷ Que el valor a buscar esté.
- ▷ Que el valor a buscar no esté.
- ▷ Que el valor a buscar esté varias veces (devuelve la primera ocurrencia).
- ▷ Que el valor a buscar esté en la primera o en la última posición.
- ▷ Que el vector esté vacío o tenga una única componente.

III.2.1.2. Búsqueda Binaria

Se aplica sobre un vector ordenado.


Algoritmo: Búsqueda Binaria

El elemento a buscar se compara con el elemento que ocupa la mitad del vector.
Si coinciden, se habrá encontrado el elemento.
En otro caso, se determina la mitad del vector en la que puede encontrarse.
Se repite el proceso con la mitad correspondiente.

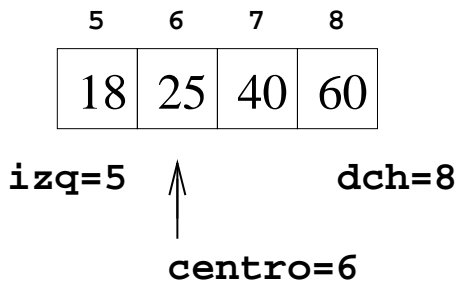
0 1 2 3 4 5 6 7 8

-8	4	5	9	12	18	25	40	60
----	---	---	---	----	----	----	----	----

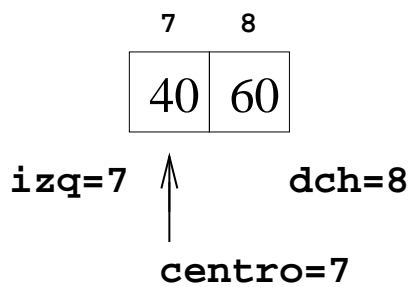
izq=0 centro=4 dch=8



40 > 12



$$40 > 25$$



Encontrado

```
int main(){
    const int TAMANIO = 50;
    char vector[TAMANIO];
    .....
    izda = 0;
    dcha = total_utilizados - 1;
    encontrado = false;

    while (izda <= dcha && !encontrado){
        centro = (izda + dcha) / 2;

        if (vector[centro] == buscado)
            encontrado = true;
        else if (buscado < vector[centro])
            dcha = centro - 1;
        else
            izda = centro + 1;
    }

    if (encontrado)
        pos_encontrado = centro;
    else
        pos_encontrado = -1;
    .....
```

http://decsai.ugr.es/jccubero/FP/III_vector_busqueda_binaria.cpp

Nota:

En el caso de que el valor a buscar estuviese repetido, la búsqueda binaria no garantiza que la posición devuelta sea la de la primera ocurrencia.

III.2.1.3. Otras búsquedas

Ejemplo. Encuentre el mínimo elemento de un vector.

$v = (h, b, c, f, d, ?, ?, ?)$

Algoritmo: Encontrar el mínimo

Inicializar mínimo a la primera componente
Recorrer el resto de componentes $v[i]$ ($i > 0$)
Actualizar, en su caso, el mínimo y la posición

```
if (total_utilizados > 0){
    minimo = vector[0];
    posicion_minimo = 0;

    for (int i = 1; i < total_utilizados ; i++){
        if (vector[i] < minimo){
            minimo = vector[i];
            posicion_minimo = i;
        }
    }
}
else
    posicion_minimo = -1;
```

Nota:

También podríamos usar `vector[posicion_minimo]`: en vez de `minimo`

http://decsai.ugr.es/jccubero/FP/III_vector_minimo.cpp

Ejemplo. Encuentre un vector dentro de otro. Este es un ejemplo de búsqueda muy importante.

```
principal = (h,b,a,a,a,b,f,d,?,?,?)  
a_buscar = (a,a,b,?,?,?, ?, ?, ?, ?)  
pos_encontrado = 3
```

Algoritmo: Buscar un vector dentro de otro

Si `a_buscar` cabe en `principal`

Recorrer `principal` -inicio- hasta que:

- `a_buscar` no quepa en lo que queda de `principal`
- se haya encontrado `a_buscar`

Recorrer las componentes de `a_buscar` -i- comparando `a_buscar[i]` con `principal[inicio + i]` hasta:

- llegar al final de `a_buscar` (se ha encontrado)
- encontrar dos caracteres distintos

```
for (int inicio = 0;
    inicio + total_utilizados_a_buscar <= total_utilizados
    &&
    !encontrado;
    inicio++){

    va_coincidiendo = true;

    for (int i = 0; i < total_utilizados_a_buscar && va_coincidiendo; i++)
        va_coincidiendo = principal[inicio + i] == a_buscar[i];

    if (va_coincidiendo){
        encontrado = true;
        pos_encontrado = inicio;
    }
}
```

http://decsai.ugr.es/jccubero/FP/III_vector_busqueda_subvector.cpp

III.2.2. Recorridos que modifican componentes

III.2.2.1. Inserción de un valor

El objetivo es insertar una componente nueva dentro del vector. Debemos desplazar una posición todas las componentes que hay a su derecha.

```
v = (t,e,c,a,?,?,?)    total_utilizados = 4
pos_insercion = 2       valor_nuevo = r
v = (t,e,r,c,a,?,?,?)  total_utilizados = 5
```

Algoritmo: Insertar un valor

```
Recorrer las componentes desde el final
hasta llegar a pos_insercion
    Asignarle a cada componente la anterior.
Colocar la nueva
```

```
for (int i = total_utilizados ; i > pos_insercion ; i--)
    vector[i] = vector[i-1];

vector[pos_insercion] = valor_nuevo;
total_utilizados++;
```

Verificación (pruebas de unidad):

- ▷ Que el vector esté vacío
- ▷ Que el elemento a insertar se sitúe el último
- ▷ Que el elemento a insertar se sitúe el primero
- ▷ Que el número de componentes utilizadas sea igual al tamaño



III.2.2.2. Eliminación de un valor

¿Qué significa eliminar/borrar una componente? Recuerde que la memoria reservada para todo el vector siempre es la misma.

Tipos de borrado:

▷ Borrado lógico.

Es el tipo de borrado usual cuando la ocupación es del tipo:

N	X	X	N	N	X	...	N
-----	-----	-----	-----	-----	-----	-----	-----

La componente se **marca** como borrada (se sustituye el valor que hubiese por N) y habrá que tenerlo en cuenta en el procesamiento posterior.

Obviamente, la ventaja es la eficiencia ya que únicamente hay que realizar una asignación. De hecho, si se prevé trabajar con datos en los que hay que hacer continuas operaciones de borrado, esta representación con huecos puede ser la recomendada.

▷ Borrado físico.

Es el tipo de borrado usual cuando la ocupación es del tipo que nos ocupa:

util = 4							
X	X	X	X	$?$	$?$...	$?$

Todas las componentes que hay a la derecha se desplazan una posición a la izquierda.

El problema es la ineficiencia. Habrá que tener especial cuidado de no realizar este tipo de borrados de forma continua.

```
v = (t,e,r,c,a,?,?,?)    total_utilizados = 5
pos_a_eliminar = 2
v = (t,e,c,a,?,?,?,?)    total_utilizados = 4
```

Algoritmo: Eliminar un valor

Recorrer las componentes desde la posición a eliminar hasta el final

Asignarle a cada componente la siguiente.

Actualizar total_utilizados

```
if (posicion >= 0 && posicion < total_utilizados){
    int tope = total_utilizados-1;

    for (int i = posicion ; i < tope ; i++)
        vector[i] = vector[i+1];

    total_utilizados--;
}
```

Los casos de prueba para la verificación serían los mismos que los del algoritmo de inserción.

III.2.3. Algoritmos de ordenación

La ordenación es un procedimiento mediante el cual se disponen los elementos de un vector en un orden especificado, tal como orden alfabético u orden numérico.

▷ **Aplicaciones:**

Mostrar los ficheros de un directorio ordenados alfabéticamente, ordenar los resultados de una búsqueda en Internet (PageRank es un método usado por Google que asigna un número de *importancia* a una página web), etc.

▷ **Técnicas de ordenación:**

- **Ordenación interna** : Todos los datos están en memoria principal durante el proceso de ordenación.
 - En la asignatura veremos métodos básicos como Inserción, Selección e Intercambio.
 - En el segundo cuatrimestre se verán métodos más avanzados como Quicksort o Mergesort.
- **Ordenación externa** : Parte de los datos a ordenar están en memoria externa mientras que otra parte está en memoria principal siendo ordenada.

▷ **Aproximaciones:**

Supondremos que los datos a ordenar están en un vector.

- Construir un segundo vector con las componentes del primero, pero ordenadas
- Modificar el vector original, cambiando de sitio las componentes. Esta aproximación es la que seguiremos.

Vamos a ver varios algoritmos de ordenación que comparten una idea común:

- ▷ El vector se dividirá en dos sub-vectores. El de la izquierda, contendrá componentes ordenadas. Las del sub-vector derecho no están ordenadas.
- ▷ Se irán cogiendo componentes del sub-vector derecho y se colocarán adecuadamente en el sub-vector izquierdo.

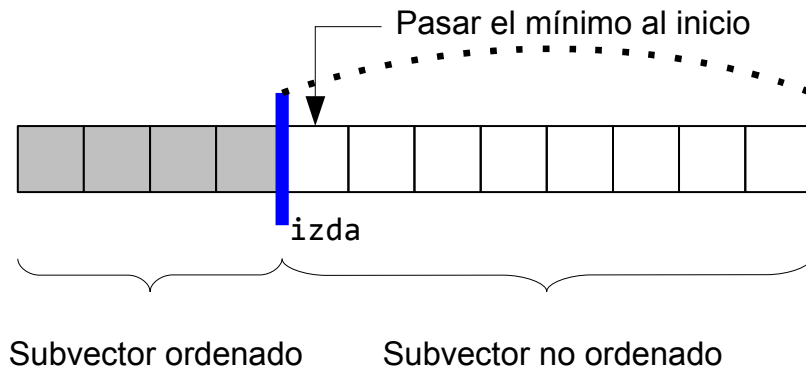
**Es muy importante conocer los métodos de ordenación
para el examen de FP**

IMPORTANT

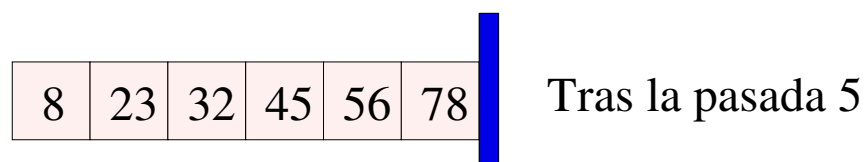
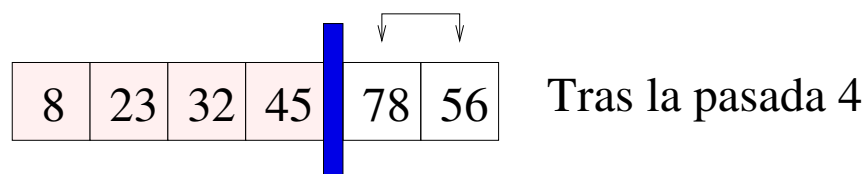
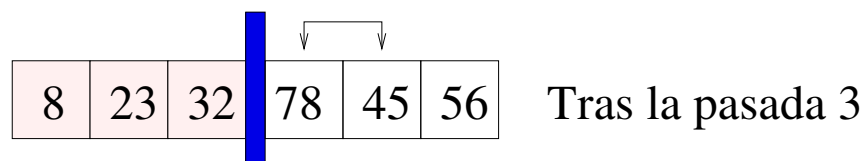
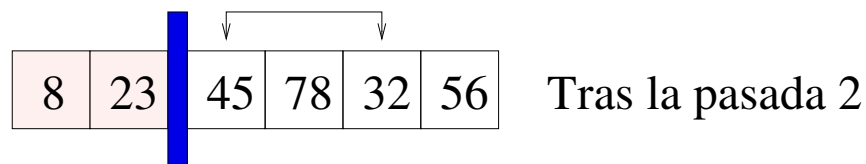
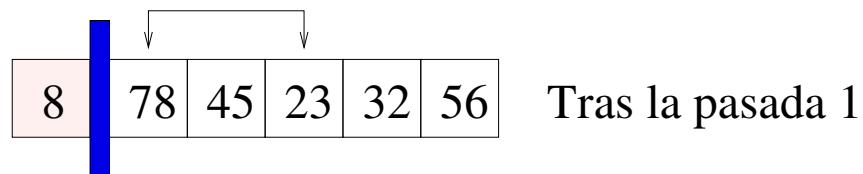
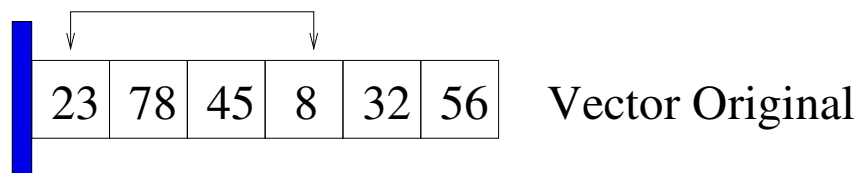
Veremos los algoritmos de ordenación por selección, inserción y burbuja.

III.2.3.1. Ordenación por Selección

Ordenación por selección (Selection sort) : En cada iteración, se selecciona la componente más pequeña del sub-vector derecho y se coloca al final del sub-vector izquierdo.



Para hacer más claro el código, en los ejemplos usaremos el tipo de dato `char` como tipo de dato base del vector y el tipo `int` para los índices que lo recorren. Obviamente, los algoritmos son idénticos si se usa cualquier otro tipo de dato base para el vector.



Algoritmo: Ordenación por Selección

Recorrer todos los elementos `v[izda]` de `v`
 Hallar la posición `pos_min` del menor elemento
 del subvector delimitado por las componentes
 `[izda , total_utilizados-1]` ;ambas inclusive!
 Intercambiar `v[izda]` con `v[pos_min]`

```
for (int izda = 0 ; izda < total_utilizados ; izda++){
    minimo = vector[izda];
    posicion_minimo = izda;

    for (int i = izda + 1; i < total_utilizados ; i++){
        if (vector[i] < minimo){
            minimo = vector[i];
            posicion_minimo = i;
        }
    }

    intercambia = vector[izda];
    vector[izda] = vector[posicion_minimo];
    vector[posicion_minimo] = intercambia;
}
```

Nota:

La última iteración (cuando `izda` es igual a `total_utilizados - 1`) no es necesaria. El bucle podría llegar hasta `total_utilizados - 2` (inclusive)

http://decsai.ugr.es/jccubero/FP/III_vector_ordenacion.cpp



Verificación (pruebas de unidad):

- ▷ **Que el vector esté vacío**
- ▷ **Que el vector sólo tenga una componente**
- ▷ **Que tenga un número de componentes par/impar**
- ▷ **Que el vector ya estuviese ordenado**
- ▷ **Que el vector ya estuviese ordenado, pero de mayor a menor**
- ▷ **Que el vector tenga todas las componentes iguales**
- ▷ **Que tenga dos componentes iguales al principio o al final o en medio**

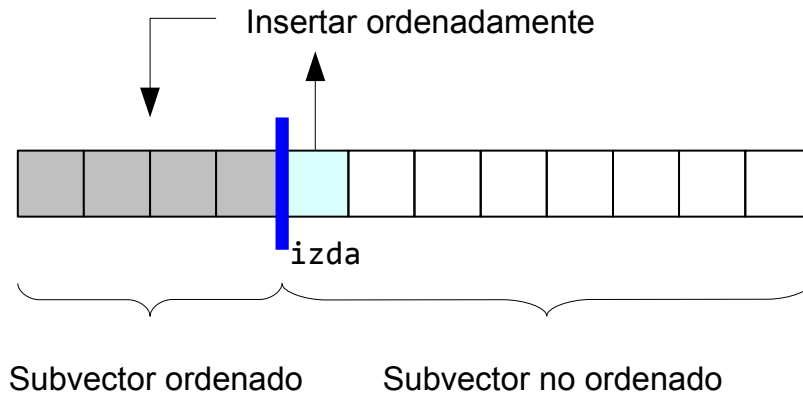
Nota. Estas pruebas también son aplicables al resto de algoritmos de ordenación

Applet de demostración del funcionamiento:

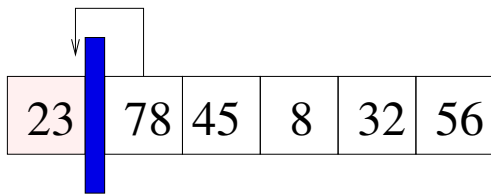
<http://www.sorting-algorithms.com/>

III.2.3.2. Ordenación por Inserción

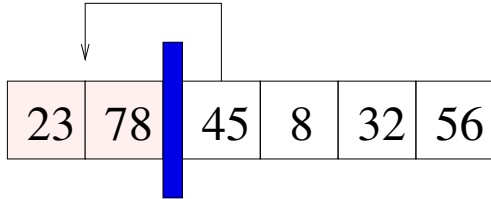
Ordenación por inserción (*Insertion sort*) : El vector se divide en dos subvectores: el de la izquierda ordenado, y el de la derecha desordenado. Cogemos el primer elemento del subvector desordenado y lo insertamos de forma ordenada en el subvector de la izquierda (el ordenado).



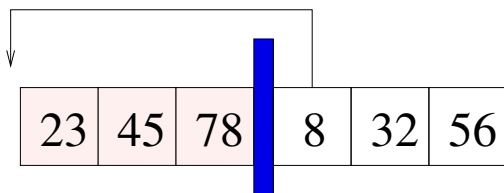
Nota. La componente de la posición *izda* (primer elemento del subvector desordenado) será reemplazada por la anterior (después de desplazar)



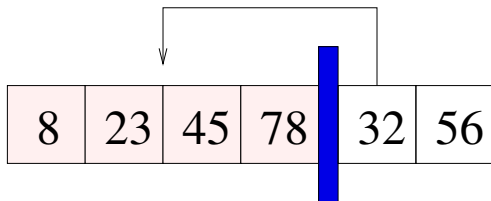
Vector Original



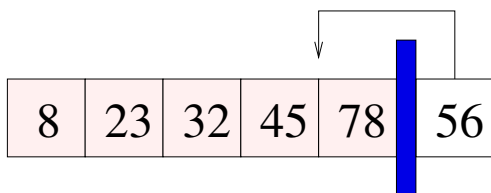
Tras la pasada 1



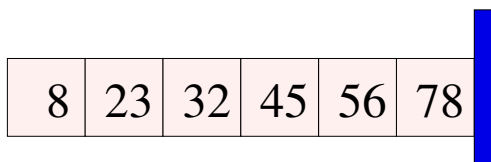
Tras la pasada 2



Tras la pasada 3



Tras la pasada 4



Tras la pasada 5

Algoritmo: Ordenación por Inserción

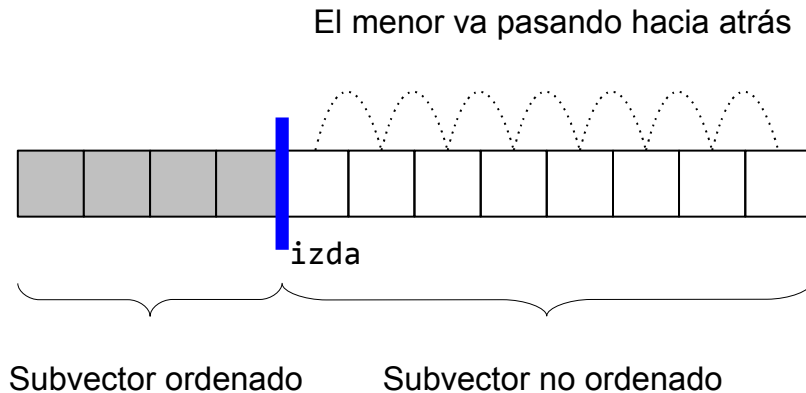
Ir fijando el inicio del subvector derecho
con un índice izda (desde 1 hasta total_utilizados - 1)
Insertar v[izda] de forma ordenada
en el subvector izquierdo

```
for (int 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;  
}
```

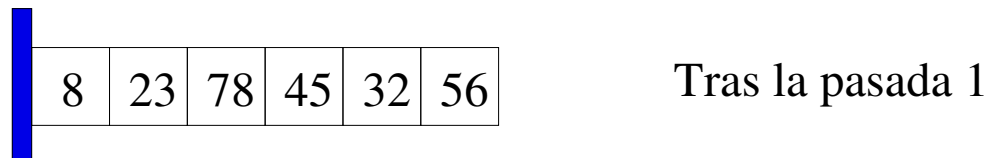
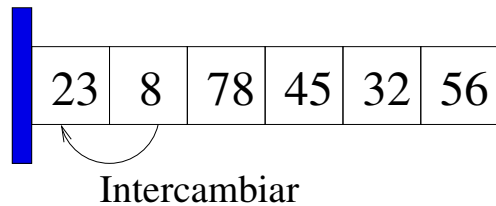
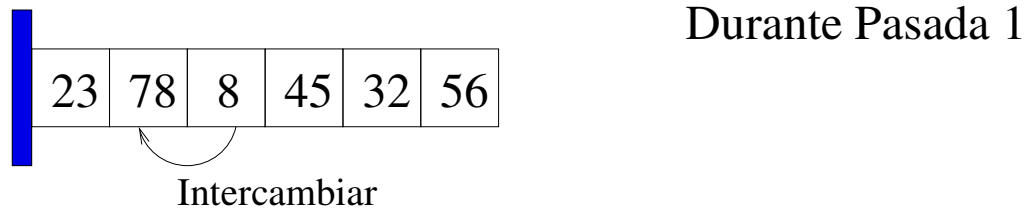
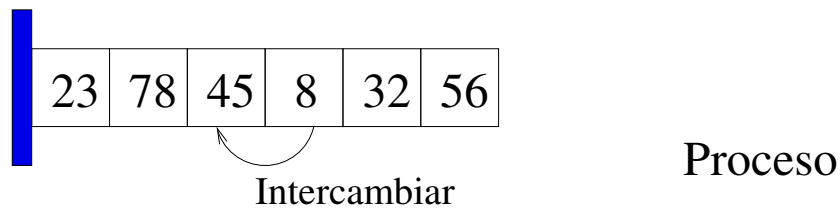
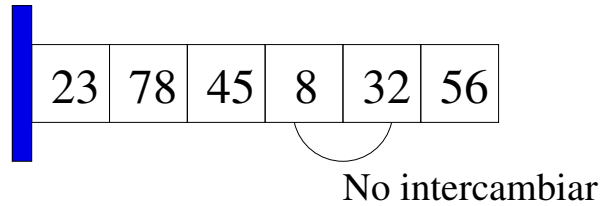
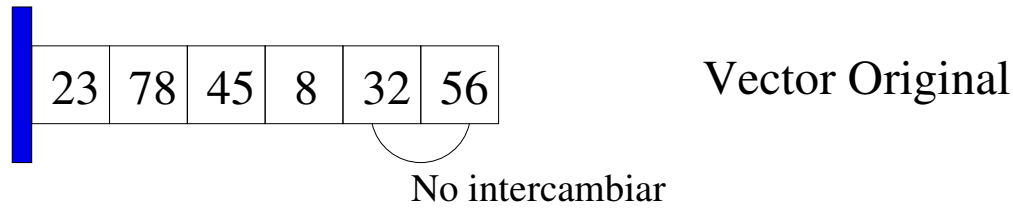
http://decsai.ugr.es/jccubero/FP/III_vector_ordenacion.cpp

III.2.3.3. Ordenación por Intercambio Directo (Método de la Burbuja)


Ordenación por intercambio directo (burbuja) (Bubble sort) : Al igual que antes, a la izquierda se va dejando un subvector ordenado. Desde el final y hacia atrás, se van comparando elementos dos a dos y se deja a la izquierda el más pequeño (intercambiándolos)



Primera Pasada




Resto de Pasadas




23	78	45	8	32	56
----	----	----	---	----	----

 Vector Original


8	23	78	45	32	56
---	----	----	----	----	----

 Tras la pasada 1


8	23	32	78	45	56
---	----	----	----	----	----

 Tras la pasada 2


8	23	32	45	78	56
---	----	----	----	----	----

 Tras la pasada 3

8	23	32	45	56	78
---	----	----	----	----	----

 Tras la pasada 4

8	23	32	45	56	78
---	----	----	----	----	----

 Tras la pasada 5

Algoritmo: Ordenación por Burbuja

```
Ir fijando el inicio del subvector derecho
con un índice izda desde 0 hasta total_utilizados - 1
  Recorrer el subvector de la derecha desde
    el final hasta el principio (izda)
    con un índice i

    Si  $v[i] < v[i-1]$  intercambiarlos
```

► **Primera Aproximación**

```
for (int izda = 0; izda < total_utilizados; izda++){
    for (int i = total_utilizados-1 ; i > izda ; i--){
        if (vector[i] < vector[i-1]){
            intercambia = vector[i];
            vector[i] = vector[i-1];
            vector[i-1] = intercambia;
        }
    }
}
```

http://decsai.ugr.es/jccubero/FP/III_vector_ordenacion.cpp

Mejora. Si en una pasada del bucle más interno no se produce ningún intercambio, el vector ya está ordenado. Lo comprobamos con una variable lógica.

► Segunda Aproximación

```
cambio = true;

for (int izda = 0; izda < total_utilizados && cambio; izda++){
    cambio = false;

    for (int i = total_utilizados-1 ; i > izda ; i--){
        if (vector[i] < vector[i-1]){
            cambio = true;
            intercambia = vector[i];
            vector[i] = vector[i-1];
            vector[i-1] = intercambia;
        }
    }
}
```

http://decsai.ugr.es/jccubero/FP/III_vector_ordenacion.cpp

Ninguno de los algoritmos de ordenación que hemos visto domina al resto, en el sentido de la eficiencia en tiempo y memoria, aunque en algunas situaciones concretas alguno puede comportarse mejor que los otros. Por ejemplo, si el vector no es muy grande y no es costosa la operación de asignación, el algoritmo de inserción suele ser el mejor. Por contra, si los datos son objetos (ver tema IV) complejos con una asignación costosa, es preferible usar la ordenación por selección (siempre que el vector también sea pequeño) En cualquier caso, podríamos decir que el método de la burbuja suele ser el que peor resultados ofrece debido a la gran cantidad de asignaciones que realiza.

"The bubble sort seems to have nothing to recommend it, except a catchy name and the fact that it leads to some interesting theoretical problems."

Donald Knuth



En otras asignaturas se verán algoritmos de ordenación más eficientes que los vistos en este tema.

III.3. Matrices

III.3.1. Declaración y operaciones con matrices

III.3.1.1. Declaración

Supongamos una finca rectangular dividida en parcelas. Queremos almacenar la producción de aceitunas, en Toneladas Métricas.

La forma natural de representar la parcelación sería usando el concepto matemático de matriz.

9.1	0.4	5.8
4.5	5.9	1.2

Para representarlo en C++ podríamos usar un vector `parcela`:

9.1	0.4	5.8	4.5	5.9	1.2
-----	-----	-----	-----	-----	-----

pero la forma de identificar cada parcela (por ejemplo `parcela[4]`) es poco intuitiva para el programador. Para resolver este problema, vamos a introducir el tipo de dato matriz.

Una matriz se declara de la forma siguiente:

`<tipo> <identificador> [<núm. filas>][<núm. columnas>;`

Como con los vectores, el tipo base de la matriz es el mismo para todas las componentes, ambas dimensiones han de ser de tipo entero, y comienzan en cero.

```
int main(){
    const int TAMANIO_FIL = 2;
    const int TAMANIO_COL = 3;

    double parcela[TAMANIO_FIL][TAMANIO_COL];
    .....
```

Otros ejemplos:

```
.....
int    imagen[MAX_ALTURA][MAX_ANCHURA];
double notas[NUM_ALUMNOS][NUM_NOTAS];
char   crucigrama [MAX_FIL][MAX_COL];
string ventas_diarias[MAX_VENTAS_POR_DIA][MAX_NUM_CODIGOS];
```

III.3.1.2. Acceso y asignación

`<identificador> [<índice fila>][<índice columna>];`

`<identificador> [<índice fila>][<índice columna>]` es una variable más del programa y se comporta como cualquier variable del tipo de dato base de la matriz.

Por ejemplo, para asignar una expresión a una celda:

`<identificador> [<índice fila>][<índice columna>] = <expresión>`

`<expresión>` ha de ser del mismo tipo de dato que el tipo base de la matriz.

Al igual que ocurre con los vectores, si los índices no son correctos, el comportamiento durante la ejecución es indeterminado (ver página 318)

Ejemplo.

```
int main(){
    const int TAMANIO_FIL = 2;
    const int TAMANIO_COL = 3;
    double parcela[TAMANIO_FIL][TAMANIO_COL];

    parcela[0][1] = 4.5; // Correcto;
    parcela[2][0] = 7.2; // Error: Fila 2 no existe.
    parcela[0][3] = 7.2; // Error: Columna 3 no existe.
    parcela[0][0] = 4;    // Correcto. Casting automático: double = int
```

III.3.1.3. Inicialización

En la declaración de la matriz se pueden asignar valores a toda la matriz. Posteriormente, no es posible: es necesario acceder a cada componente independientemente.

La forma *segura* es poner entre llaves los valores de cada fila.

```
int parc[2][3] = {{1,2,3},{4,5,6}};    // parc tendrá: 1 2 3
                                           //              4 5 6
```

Si no hay suficientes inicializadores para una fila determinada, los elementos restantes se inicializan a 0.

```
int parc[2][3] = {{1},{3,4,5}};    // parc tendrá: 1 0 0
                                           //              3 4 5
```

Si se eliminan las llaves que encierran cada fila, se inicializan los elementos de la primera fila y después los de la segunda, y así sucesivamente.

```
int parc[3][4] = {1, 2, 3, 4, 5}; // parc tendrá: 1 2 3 4
                                           //              5 0 0 0
                                           //              0 0 0 0
```


III.3.1.4. Representación en memoria (Ampliación)

Todas las posiciones de una matriz están realmente contiguas en memoria. La representación exacta depende del lenguaje. En C++ se hace por filas:



m

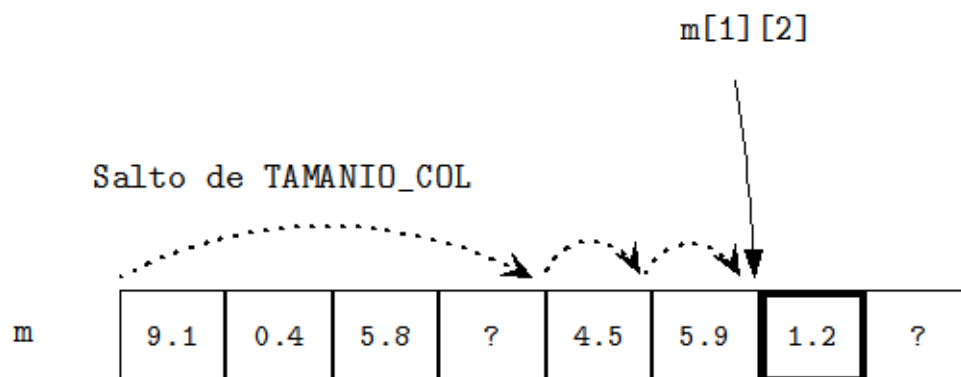
9.1	0.4	5.8	?
4.5	5.9	1.2	?

m

9.1	0.4	5.8	?	4.5	5.9	1.2	?
-----	-----	-----	---	-----	-----	-----	---

m contiene la dirección de memoria de la primera componente.

Para calcular dónde se encuentra la componente $m[1][2]$ el compilador debe *pasar a la segunda fila*. Lo consigue trasladándose tantas posiciones como diga `TAMANIO_COL`, a partir del comienzo de **m**. Una vez ahí, *salta 2 posiciones y ya está en $m[1][2]$* .



Conclusión: Para saber dónde está $m[i][j]$, el compilador necesita saber cuánto vale `TAMANIO_COL`, pero no `TAMANIO_FIL`. Para ello, da tantos saltos como indique la expresión $i * \text{TAMANIO_COL} + j$

III.3.1.5. Más de dos dimensiones

Podemos declarar tantas dimensiones como queramos. Sólo es necesario añadir más corchetes.

Por ejemplo, para representar la producción de una finca dividida en 2×3 parcelas, y dónde en cada parcela se practican cinco tipos de cultivos, definiríamos:

```
const int DIV_HOR = 2,  
        DIV_VERT = 3,  
        TOTAL_CULTIVOS = 5;  
  
double parcela[DIV_HOR][DIV_VERT][TOTAL_CULTIVOS];  
  
// Asignación al primer cultivo de la parcela 1,2  
  
parcela[1][2][0] = 4.5;
```

El tamaño (número de componentes) de una matriz es el producto de los tamaños de cada dimensión. En el ejemplo anterior, se han reservado en memoria un total de $2 \times 3 \times 5 = 30$ enteros.

III.3.2. Gestión de componentes útiles con matrices

¿Qué componentes usamos en una matriz? Depende del problema.

En este apartado trabajamos directamente en el `main`. En el próximo tema lo haremos dentro de una clase.

Veamos los tipos de gestión de componentes no utilizadas más usados.

III.3.2.1. Se usan todas las componentes

Ocupamos todas las componentes reservadas. Las casillas no utilizadas tendrán un valor especial *N* del tipo de dato de la matriz.

`MAX_FIL = 15, MAX_COL = 20`

<i>N</i>	<i>X</i>	<i>N</i>	<i>N</i>	<i>X</i>	<i>N</i>	...	<i>X</i>
<i>X</i>	<i>X</i>	<i>N</i>	<i>N</i>	<i>N</i>	<i>X</i>	...	<i>N</i>
<i>N</i>	<i>N</i>	<i>X</i>	<i>X</i>	<i>X</i>	<i>N</i>	...	<i>N</i>
...
<i>X</i>	<i>N</i>	<i>X</i>	<i>N</i>	<i>X</i>	<i>N</i>	...	<i>X</i>

Ejemplo. Queremos gestionar un crucigrama que siempre tiene un tamaño fijo 5 x 6 . El carácter # representa un separador entre palabras.

```
A C E R O S
L A S E R #
T I O # D E
A D # N E D
R A Z O N #
```

Leemos los datos e imprimimos el crucigrama.

```
#include <iostream>
using namespace std;

int main(){
    const int MAX_FIL = 5, MAX_COL = 6;
    char crucigrama [MAX_FIL][MAX_COL];

    for (int i = 0; i < MAX_FIL; i++)
        for (int j = 0; j < MAX_COL; j++)
            cin >> crucigrama[i][j];

    for (int i = 0; i < MAX_FIL; i++){
        for (int j = 0; j < MAX_COL; j++)
            cout << crucigrama[i][j] << " ";
        cout << "\n";
    }
}
```

Los datos deben introducirse con todos los caracteres consecutivos:

```
ACEROSLASER#TIO#DEAD#NEDRAZON#
```

http://decsai.ugr.es/jccubero/FP/III_crucigrama_base.cpp

Ejemplo. Almacenamos una imagen en blanco y negro en una matriz de pixeles: cada uno es un entero positivo que representa el nivel de gris. Se supone que tiene pixeles incorrectos dados por un número negativo.

Queremos sustituir cada uno de esos valores incorrectos por la media aritmética de sus 8 vecinos más cercanos. En el cómputo de la media no intervienen los posibles pixeles incorrectos que hubiese. Los pixeles del borde se dejan como estuviesen.

Así pues, si la imagen inicial fuese la siguiente:

-1	-1	-7	1	9	1
-2	-2	-1	8	-1	1
-6	-1	-3	1	3	1
-1	-2	-5	4	4	4
1	5	5	5	5	1

la imagen modificada sería:

-1	-1	-1	1	9	1
-2	0	3	8	3	1
-1	0	4	1	3	1
-1	3	4	4	4	4
1	5	5	5	5	1

Debemos recorrer todas las filas y columnas de la imagen original. Lo hacemos con un par de variables `fil`, `col`. Para cada casilla `fil`, `col`, debemos recorrer sus 8 vecinos. Lo hacemos con dos variables `i`, `j`:

```
for (fil = 1 ; fil < util_fil - 1; fil++){
    for (col = 1; col < util_col - 1; col++){

        if (imagen[fil][col] < 0){                                // (*)
            media = 0;
            num_vecinos_correctos = 0;

            for (i = fil - 1; i <= fila + 1; i++){
                for (j = col - 1; j <= col + 1; j++){
                    if (! (i == fil && j == col)){                // (**)
                        if (imagen[i][j] >= 0){                  // (***)
                            media = media + imagen[i][j];
                            num_vecinos_correctos++;
                        }
                    }
                }
                media = media / num_vecinos_correctos;
                imagen_suavizada[fil][col] = media;
            }
        }
    }
}
```

Algunas cuestiones:

- ▷ **La condición (**) controla que no se considere como vecino de `fil`, `col` la misma casilla `fil`, `col`. Nos lo podríamos haber ahorrado ya que tenemos la garantía de que `imagen[fil][col]` es un valor negativo (*), por lo que la condición (***) será falsa y no corremos el peligro de computar el valor en la media.**

Sin embargo, el código queda más claro dejando la condición () y no depende de posibles cambios de criterios en el futuro.**

- ▷ **Hay cierto código duplicado ya que la comprobación de si un pixel**

es correcto se realiza en dos sitios: (*) y (). Lo correcto sería empaquetar ese código en un módulo. Lo veremos en el tema IV.**

- ▷ **Falta asignar los valores del borde de la imagen suavizada. Consulte la solución completa en el siguiente enlace:**

http://decsai.ugr.es/jccubero/FP/III_ImagenSuavizada.cpp

III.3.2.2. Se ocupan todas las columnas pero no todas las filas (o al revés)

Debemos estimar el máximo número de filas que podamos manejar (MAX_FIL) para reservar memoria suficiente. Usaremos una variable `filas_utilizadas` para saber cuántas se usan en cada momento de la ejecución del programa.

```
MAX_FIL = 15, MAX_COL = 20
filas_utilizadas = 3
```

X	X	X	X	X	X	...	X
X	X	X	X	X	X	...	X
X	X	X	X	X	X	...	X
?	?	?	?	?	?	...	?
...
?	?	?	?	?	?	...	?

Ejemplo. Queremos gestionar las notas de los alumnos de una clase. Cada alumno tiene 3 notas de evaluación continua, dos notas de exámenes de prácticas y una nota del examen escrito.

```
MAX_ALUMNOS = 100, NUM_NOTAS = 6
num_alumnos = 2
```

4.0	7.5	8.0	6.5	7.0	4.0
3.0	2.5	4.0	3.0	2.5	1.5
?	?	?	?	?	?
...
?	?	?	?	?	?

Calculamos la nota media de cada alumno, aplicando unas ponderaciones.

```
int main(){
    const int MAX_ALUMNOS = 100, NUM_NOTAS = 6;
    double notas[MAX_ALUMNOS][NUM_NOTAS];
    int num_alumnos;
    double nota_final;
    double ponderacion[NUM_NOTAS] = {0.1/3.0, 0.1/3.0, 0.1/3.0,
                                      0.05, 0.15, 0.7};

    cout.precision(3);

    do{
        cin >> num_alumnos;
    }while (num_alumnos < 0);

    for (int i = 0; i < num_alumnos; i++)
        for (int j = 0; j < NUM_NOTAS; j++)
            cin >> notas[i][j];

    for (int i = 0; i < num_alumnos; i++){
        nota_final = 0;

        for (int j = 0; j < NUM_NOTAS; j++)
            nota_final = nota_final + notas[i][j] * ponderacion[j];

        cout << "\nNota final del alumno número " << i
              << " = " << nota_final;
    }
}
```

http://decsai.ugr.es/jccubero/FP/III_notas_base.cpp

III.3.2.3. Se ocupa un bloque rectangular

Ocupamos un bloque completo. Lo normal será la esquina superior izquierda.

Por cada dimensión debemos estimar el máximo número de componentes que podamos almacenar y usaremos tantas variables como dimensiones haya para saber cuántas componentes se usan en cada momento.

Este tipo de ocupación es bastante usual, aunque hay que tener en cuenta que podemos estar desperdiciando mucha memoria reservada.

```
MAX_FIL = 15, MAX_COL = 20
util_fil = 2, util_col = 4
```

X	X	X	X	?	?	...	?
X	X	X	X	?	?	...	?
?	?	?	?	?	?	...	?
...
?	?	?	?	?	?	...	?

Ejemplo. Queremos gestionar un crucigrama de cualquier tamaño.

```
A  C  E  R  O  S  ?  ...  ?
L  A  S  E  R  #  ?  ...  ?
T  I  O  #  D  E  ?  ...  ?
A  D  #  N  E  D  ?  ...  ?
R  A  Z  O  N  #  ?  ...  ?
?  ?  ?  ?  ?  ?  ?  ...  ?
...
?  ?  ?  ?  ?  ?  ?  ...  ?
```

Leemos los datos del crucigrama y los imprimimos.

```
#include <iostream>
using namespace std;

int main(){
    const char SEPARADOR = '#';
    const int MAX_FIL = 5, MAX_COL = 6;
    char crucigrama [MAX_FIL][MAX_COL];
    int util_fil, util_col;

    do{
        cin >> util_fil;
    }while (util_fil > MAX_FIL || util_fil < 0);

    do{
        cin >> util_col;
    }while (util_col > MAX_COL || util_col < 0);

    for (int i = 0; i < util_fil; i++)
        for (int j = 0; j < util_col; j++)
            cin >> crucigrama[i][j];

    for (int i = 0; i < util_fil; i++){
        for (int j = 0; j < util_col; j++)
            cout << crucigrama[i][j] << " ";
        cout << "\n";
    }
}
```

Ejercicio. Un problema más complejo: Obtener un vector de `string` con las palabras del crucigrama (cada palabra será un `string`)

http://decsai.ugr.es/jccubero/FP/III_crucigrama_extrae_palabras.cpp

Ejemplo. Queremos gestionar una sopa de letras de cualquier tamaño (no hay carácter # separador de palabras como en un crucigrama).

```
A C E R O S ? ... ?
L A S E R I ? ... ?
T I O B D E ? ... ?
A D I N E D ? ... ?
R A Z O N P ? ... ?
? ? ? ? ? ? ? ... ?
...
? ? ? ? ? ? ? ... ?
```

Supongamos que queremos buscar una palabra (en horizontal y de izquierda a derecha). Usamos como base el algoritmo de búsqueda de un vector dentro de otro visto en la página **337**

Algoritmo: Buscar una palabra en una sopa de letras

Recorrer todas las filas -fil- hasta terminarlas
o hasta encontrar la palabra

Con una fila fija, recorrer sus columnas -col_inicio-
hasta que:

- a_buscar no quepa en lo que queda de la fila
- se haya encontrado a_buscar

Recorrer las componentes de a_buscar -i- comparando
a_buscar[i] con sopa[fil][col_inicio + i] hasta:

- llegar al final de a_buscar (se ha encontrado)
- encontrar dos caracteres distintos

```
#include <iostream>
using namespace std;

int main(){
    const int MAX_FIL = 30, MAX_COL = 40;
    char sopa [MAX_FIL][MAX_COL];
    int util_fil, util_col;

    char a_buscar [MAX_COL];
    int  tamaño_a_buscar;
    bool encontrado, va_coincidiendo;
    int  fil_encontrado, col_encontrado;

    do{
        cin >> util_fil;
    }while (util_fil > MAX_FIL || util_fil < 0);

    do{
        cin >> util_col;
    }while (util_col > MAX_COL || util_col < 0);

    for (int i = 0; i < util_fil; i++)
        for (int j = 0; j < util_col; j++)
            cin >> sopa[i][j];

    cin >> tamaño_a_buscar;

    for (int i = 0; i < tamaño_a_buscar; i++)
        cin >> a_buscar[i];

    encontrado = false;
    fil_encontrado = col_encontrado = -1;
```

```
for (int fil = 0; fil < util_fil && !encontrado; fil++){

    for (int col_inicio = 0;
        col_inicio + tamaño_a_buscar <= util_col && !encontrado;
        col_inicio++){

        va_coincidiendo = true;

        for (int i = 0; i < tamaño_a_buscar && va_coincidiendo; i++)
            va_coincidiendo = sopa[fil][col_inicio + i] == a_buscar[i];

        if (va_coincidiendo){
            encontrado = true;
            fil_encontrado = fil;
            col_encontrado = col_inicio;
        }
    }
}

if (encontrado)
    cout << "\nEncontrado en " << fil_encontrado << "," << col_encontrado;
else
    cout << "\nNo encontrado";
}
```

http://decsai.ugr.es/jccubero/FP/III_sopa.cpp

III.3.2.4. Se ocupan las primeras filas, pero con tamaños distintos

Debemos usar un vector para almacenar el tamaño actual de cada fila.

MAX_FIL = 15, MAX_COL = 20 , util_fil = 3, util_col = {5, 2, 4}

X	X	X	X	X	?	...	?
X	X	?	?	?	?	...	?
X	X	X	X	?	?	...	?
?	?	?	?	?	?	...	?
...
?	?	?	?	?	?	...	?

Ejemplo. Almacenamos las notas de evaluación continua de los alumnos. Cada alumno se corresponde con una fila y el número de notas puede variar entre los distintos alumnos, pero no serán más de 8.

MAX_ALUMNOS = 100, MAX_NUM_NOTAS = 8
num_alumnos = 2 , num_notas_por_alumno = {4, 3, ?, ..., ?}

9.5	8.3	9.1	6.7	?	?	...	?
3.4	4.1	2.3	?	?	?	...	?
?	?	?	?	?	?	...	?
...
?	?	?	?	?	?	...	?

```
int main(){
    const int MAX_ALUMNOS = 100, MAX_NUM_NOTAS = 8;
    double notas_ev_continua [MAX_ALUMNOS][MAX_NUM_NOTAS];
    int num_notas_por_alumno[MAX_ALUMNOS];          int num_alumnos;
```

Una consideración final importante:

Hemos de ser conscientes de que el número de componentes no utilizadas en una matriz puede ser muy grande. Por lo tanto, en matrices de gran tamaño, se hace necesario usar otras técnicas de almacenamiento de datos con huecos, ya que el número de datos desperdiciados puede ser demasiado elevado.

La solución pasará por usar **memoria dinámica (dynamic memory)** . Se verá en el segundo cuatrimestre.

Bibliografía recomendada para este tema:

Una referencia bastante adecuada es Wikipedia (pero la versión en inglés)

http://en.wikipedia.org/wiki/Category:Search_algorithms

http://en.wikipedia.org/wiki/Category:Sorting_algorithms

Resúmenes:

En C++ no se puede declarar un vector con un tamaño variable (En C99 sí se puede)

Las componentes de un vector son datos como los vistos hasta ahora, por lo que le serán aplicables las mismas operaciones definidas por el tipo de dato asociado.

Para aumentar la eficiencia, el compilador no comprueba que el índice de acceso a las componentes esté en el rango correcto, por lo que cualquier acceso de una componente con un índice fuera de rango tiene consecuencias imprevisibles.

No **añada** caracteres a un string mediante el acceso directo a la componente con la notación corchete. Use `push_back` en su lugar.

Hemos de ser conscientes de que el número de componentes no utilizadas en una matriz puede ser muy grande. Por lo tanto, en matrices de gran tamaño, se hace necesario usar otras técnicas de almacenamiento de datos con huecos, ya que el número de datos desperdiciados puede ser demasiado elevado.

La solución pasará por usar **memoria dinámica (dynamic memory)** . Se verá en el segundo cuatrimestre.

Consejo: *Fomente el uso de constantes en vez de literales para especificar el tamaño de los vectores.*



Es muy importante conocer los métodos de ordenación para el examen de FP



Tema IV

Funciones y Clases

Objetivos:

- ▷ Introducir el concepto de función como una herramienta esencial que nos permitirá empaquetar un conjunto de instrucciones.
- ▷ Introducir el principio de ocultación de información.
- ▷ Introducir el principio de encapsulación que permitirá empaquetar datos e instrucciones en un mismo módulo: la Clase.
- ▷ Construir clases sencillas que cumplan principios básicos de diseño.

IV.1. Funciones

IV.1.1. Fundamentos

IV.1.1.1. Las funciones realizan una tarea

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.

El objetivo de una función es empaquetar un conjunto de instrucciones para resolver una tarea concreta. Las funciones como `sqrt`, `tolower`, etc., no son sino ejemplos de funciones incluidas en `cmath` y `cctype`, respectivamente.

Las funciones reciben una información de entrada especificada por unos parámetros y devuelven un valor.

Ejemplo. Supongamos que tenemos que 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 una 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);
```

IV.1.1.2. Definición

```
<tipo> <nombre-función> (<parámetros 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.

En el segundo cuatrimestre se verá cómo relajar esta restricción.

- ▷ Diremos que

<tipo> <nombre-función> (<parámetros formales>)

es la ***cabecera (header)*** de la función.

Los parámetros formales son los datos que la función necesita conocer del exterior, para poder hacer sus cálculos

- ▷ 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.

El valor calculado por la función es el resultado de evaluar la expresión en la sentencia `return`

- ▷ La llamada a una función constituye 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. Si es un nombre compuesto, también irá en mayúscula la primera letra. Es el denominado estilo *UpperCamel-Case* (también conocido como *PascalCase*).

IV.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. Si hay más de uno, se separan con comas.

Los parámetros formales sólo se conocen dentro de la función.

Si no hay ningún parámetro, basta poner `()` o, si se prefiere, `(void)`

- ▷ 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> (<parámetros actuales>)`

En la llamada no se especifica el tipo. Si hay más de uno, también se separan con comas.

Si no hay ningún parámetro, la llamada será `<nombre-función>()`

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.

Ejemplo. Construir una función para calcular el cuadrado de un número.

```
double Cuadrado(double entrada){    // entrada: parámetro formal
    return entrada*entrada;
}
int main(){
    double resultado, valor;

    valor = 4;
    resultado = Cuadrado(valor);    // valor: parámetro actual
                                    // Cuadrado(valor) es una expresión re
    cout << "El cuadrado de " << valor << " es " << resultado;
}
```

► Flujo de control

Cuando se ejecuta la llamada a la función, 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*:

$$\langle \text{parámetro formal} \rangle = \langle \text{parámetro actual} \rangle$$

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 ésta 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.

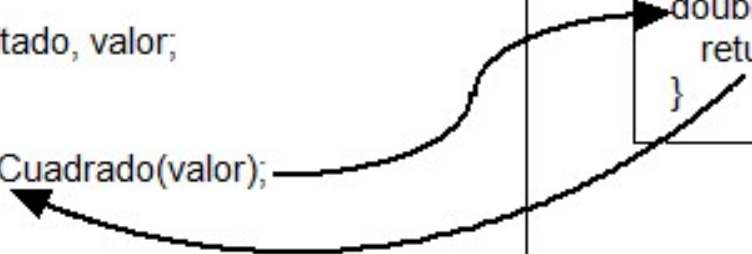
Siguiendo el ejemplo de la función Cuadrado:

```
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

```
int main(){  
    double resultado, valor;  
  
    valor = 4;  
    resultado = Cuadrado(valor);  
  
    cout << "El cuadrado de " << valor << " es "  
    << resultado;  
}
```

```
double Cuadrado(double entrada){  
    return entrada*entrada;  
}
```



► 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; // Valores indeterminados: ?, ?
    resultado = Cuadrado(valor); // Devuelve un valor indeterminado
                                // Se ha producido un 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);
```

En la medida de lo posible, no abusaremos de este tipo de llamadas.

- ▷ 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.

Ejercicio. Defina la función `Hipotenusa`. Para calcular el cuadrado de un cateto, llame a la función `Cuadrado`.

http://decsai.ugr.es/jccubero/FP/IV_hipotenusa.cpp

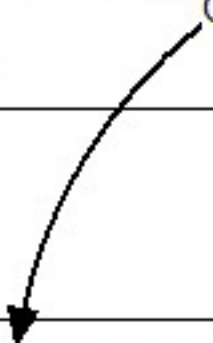
Flujo de control:

```
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 = 4.0;

    resultado = Cuadrado(valor);

    // resultado se queda con 16.0
    // valor sigue siendo 4.0
    .....
}
```

Ejemplo. Compruebe 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/jccubero/FP/IV_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, se produce un **comportamiento indeterminado** (*undefined behaviour*) (recuerde lo visto en la página 318).*

Ejercicio. Comprobar si dos reales son iguales, aceptando un margen en la diferencia de 0.000001

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***

Si en vez de usar una función simplemente copiamos y pegamos el código, cualquier cambio de los datos concretos que aparecen en dicho código nos obliga a cambiar todas y cada una de las apariciones de dichos datos, por lo que podríamos olvidar alguna y cometer un error lógico.

▷ ***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

IV.1.1.4. Datos locales

Recuerde que el *ámbito (scope)* de un dato (variable o constante) v es el conjunto de todos aquellos sitios que pueden acceder a v .

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)*.

Usaremos los datos locales en dos situaciones:

1. Es usual declarar un dato local asociado al valor que devolverá la función. Se le asigna un valor y se devuelve al final de la función con una sentencia `return`. Usaremos un nombre similar a la función pero en minúscula.
2. Para almacenar resultados temporales necesarios para hacer los cálculos requeridos en la función.

De hecho, los parámetros formales se pueden considerar datos locales.

```
<tipo> <nombre-función> (<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. Calcule 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 variables `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 << "C  mputo del factorial de un n  mero\n";
    cout << "\nIntroduzca un entero: ";
    cin >> valor;

    resultado = Factorial(valor);
    cout << "\nFactorial de " << valor << " = " << resultado;
}
```

http://decsai.ugr.es/jccubero/FP/IV_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, o con los de cualquier dato de cualquier otra función: son datos distintos.

```
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.

```
.....  
double Hipotenusa(double un_lado, double otro_lado){  
    return sqrt(Cuadrado(un_lado) + Cuadrado(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;  
}
```

Ejemplo. Compruebe si un número es primo (recuerde el algoritmo visto en la página 243)

```
bool EsPrimo(int valor){
    bool es_primo;

    es_primo = true;

    for (int 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/jccubero/FP/IV_primo.cpp

Ejemplo. Calcule 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/jccubero/FP/IV_mcd_funcion.cpp

Ejemplo. Construya una función para leer un entero estrictamente 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: ");
    .....
}
```

La lectura anterior no nos protege de desbordamientos. Para ello, habría que realizar la lectura del número sobre una cadena de caracteres y analizar dicha cadena.

IV.1.1.5. 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.

¿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 **538**

IV.1.1.6. 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;
}
```

Ejercicio. Cambiar la implementación de a función `Max3` para que llame a una función `Max` que calcule el máximo de dos enteros.

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.

Ampliación:

Consulte el capítulo 19 del libro Code Complete de Steve McConnell sobre normas para escribir comentarios claros y fáciles de mantener.



IV.1.1.7. 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. Construya una función para calcular el número de combinaciones sin repetición y sin importar el orden. La solución la da la fórmula del número combinatorio. El combinatorio de dos enteros a y b se define como:

$$\binom{a}{b} = \frac{a!}{b!(a-b)!}$$

<http://www.disfrutalasmaticas.com/combinatoria/combinaciones-permutaciones.html>

Construimos la función `Combinatorio` **que llama a la función** `Factorial`:

```
#include <iostream>
using namespace std;

// Prec: 0 <= n <= 20
long long Factorial (int n){
    int i;
    long long fact = 1;

    for (i = 2; i <= n; i++)
        fact = fact * i;

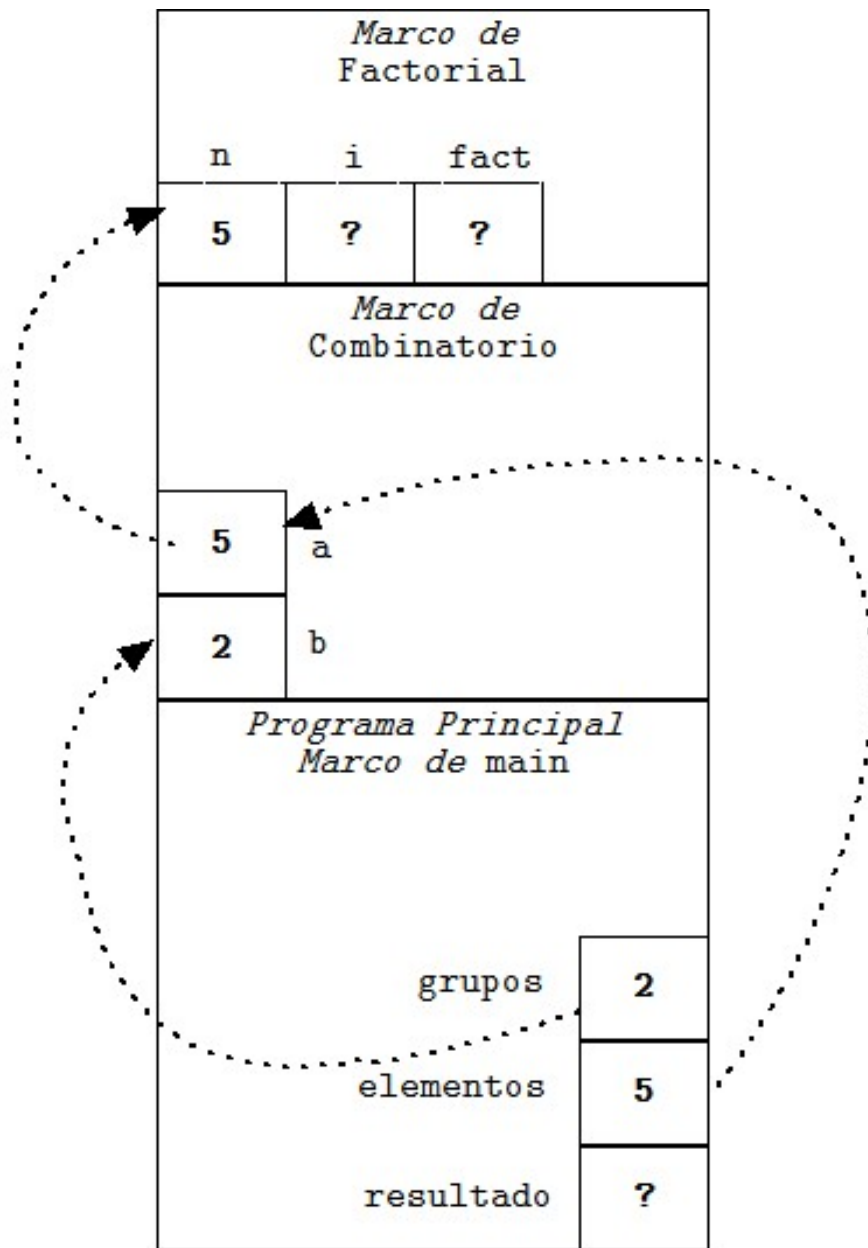
    return fact;
}

// Prec: 0 <= a, b <= 20
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;  
}
```

http://decsai.ugr.es/jccubero/FP/IV_combinatorio.cpp



Pila

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.

Nota:

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, se devuelve 0 por defecto, por lo que podríamos suprimir **return 0;** (sólo en la función **main**)

IV.1.1.8. Funciones void

Hay situaciones en las que queremos englobar un conjunto de sentencias para resolver una tarea pero no queremos devolver un valor. En este caso construiremos un tipo especial de función. Los lenguajes suelen referirse a este tipo de funciones con el nombre de *procedimiento (procedure)*. En C++ se denominan 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 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> (<parámetros formales>) {  
    <constantes locales>  
    <variables locales>  
    <sentencias>  
}
```

Algunas consideraciones:

- ▷ **void puede considerarse como un *tipo de dato vacío*.**
- ▷ **Obsérvese que no hay sentencia `return`. La función `void` termina cuando se ejecuta la última sentencia de la función**

Nota:

Realmente, el lenguaje permite incluir una sentencia `return` ; en cualquier sitio para finalizar la ejecución del `void`, pero, como ya comentaremos en la página 445, debemos evitar esta forma de terminar una 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){  
    .....  
}  
int main(){  
    int parametro_actual;  
    .....  
    MiFuncionVoid(parametro_actual);  
    .....
```

```
#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.

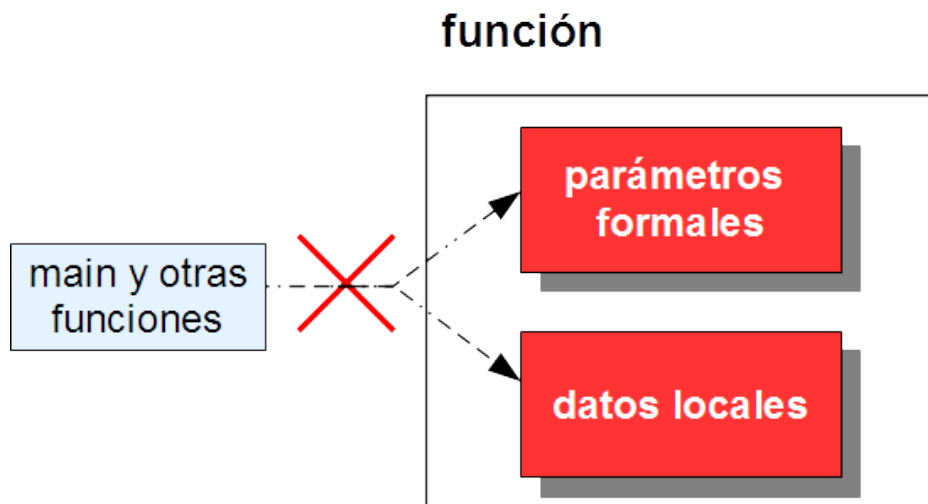
Ejercicio. Aisle la impresión de las líneas de asteriscos en una función `ImprimeLineas` y cambie la definición de la función `Presentacion` para que llame a esta nueva función. Al aislar la tarea de imprimir líneas en la función `ImprimeLineas`, estamos fomentando su reutilización en otros programas y funciones, tal y como recomendaremos en la página 422.

IV.1.1.9. 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 el código que hay en 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 400) podemos mejorar la implementación, parando al llegar a la raíz cuadrada del valor sin haber encontrado un divisor (ver página 244)

```
bool EsPrimo(int valor){
    bool es_primo;
    double tope;

    es_primo = true;
    tope = sqrt(valor);

    for (int 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 409. Para calcular los resultados posibles de la lotería primitiva, habría que calcular el número combinatorio 45 sobre 6, es decir, `elementos = 45` y `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:

```
// Prec: 1 <= b <= 20,
      a >= b,
      a(a-1)...(a-b+1) <= numeric_limits<long long>::max()
long long Combinatorio(int a, int b){
    long long 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/jccubero/FP/IV_combinatorio.cpp

En resumen:

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.

IV.1.2. Diseño de funciones

IV.1.2.1. Funciones genéricas vs funciones específicas

A la hora de resolver un problema, debemos descomponerlo en subproblemas más simples e intentar reutilizar funciones que ya hayamos construido previamente.

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)

Si reutilizamos la función definida en la página 419, bastaría incluirla en nuestro código. Podríamos añadir una función específica para este programa, a saber, la función `NumPartidos`:

```
long long Combinatorio(int a, int b){
    long long 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 NumPartidos(int num_equipos) {
    return Combinatorio(num_equipos, 2);
}
```

```
}

int main(){
    // Gestión de partidos
    .....
}
```

Por un lado tenemos la función genérica `Combinatorio` que seguramente será reutilizada en muchos otros programas. Por otro lado, tenemos la función `NumPartidos` que posiblemente no se reutilice en ningún otro programa, pero ayuda a organizar el programa de gestión de partidos.

Cuando construya las funciones que resuelven tareas concretas de un programa, procure diseñarlas de tal forma que se favorezca su reutilización en otros programas.

No todas las funciones se reutilizarán en otros contextos: habrá que encontrar un equilibrio entre funciones más genéricas y funciones más específicas de una solución concreta.

IV.1.2.2. ¿Cuántos parámetros pasamos?

¿Qué es mejor: muchos parámetros o pocos? Los justos y necesarios.

Ejemplo. Faltan parámetros:

```
.....  
long long Factorial(){  
    long long fact = 1;  int n;  
  
    cin >> n;  
  
    for (int i = 2; i <= n; i++)  
        fact = fact * i;  
  
    return fact;  
}  
  
int main(){  
    long long resultado;  
  
    resultado = Factorial();  
    cout << "Factorial = " << 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.

```
#include <iostream>
using namespace std;

long long Factorial (int i, int n){
    long long fact = 1;

    while (i <= n){
        fact = fact * i;
        i++;
    }

    return fact;
}

int main(){
    int n = 5;
    long long resultado;
    int que_pinta_esta_variable_en_el_main = 1;

    // Resultado correcto:
    resultado = Factorial(que_pinta_esta_variable_en_el_main, n);

    // Resultado incorrecto:
    resultado = Factorial(4, n);
    .....
}
```



Observe que el correcto funcionamiento de esta versión de la función Factorial depende de que se le pase como primer parámetro un valor concreto, siempre el mismo (1). Esto no es admisible.

Ejemplo. Demasiados parámetros. Construya una función para calcular la suma de los divisores de un entero (algoritmo visto en la página 282) La siguiente versión solucionaría correctamente el problema planteado:

```
int SumaDivisores (int valor){  
    int suma = 0;  
    int ultimo_divisor = valor/2;  
  
    for (int divisor = 2; divisor <= ultimo_divisor; divisor++) {  
        if (valor % divisor == 0)  
            suma = suma + divisor;  
    }  
  
    return suma;  
}
```



La llamada sería:

```
resultado = SumaDivisores(10); // Resultado siempre correcto
```

Sin embargo, la siguiente versión no sería correcta ya que exige que se le pase un segundo parámetro, totalmente innecesario:

```
int SumaDivisores (int valor, int ultimo_divisor){  
    int suma = 0;  
  
    for (int divisor = 2; divisor <= ultimo_divisor; divisor++) {  
        if (valor % divisor == 0)  
            suma = suma + divisor;  
    }  
    return suma;  
}
```



La llamada sería:

```
resultado = SumaDivisores(10, 5); // Resultado correcto  
resultado = SumaDivisores(10, 4); // Resultado incorrecto
```

Ejemplo. Demasiados parámetros:

Re-escribimos el ejemplo del MCD de la página 401. 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;

// Resultado correcto:
maximo_comun_divisor = MCD(un_entero, otro_entero, menor);

// Resultado incorrecto:
maximo_comun_divisor = MCD(un_entero, otro_entero, 9);
```

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:

```
resultado = MCD(4, 8, 4); // Resultado correcto
resultado = MCD(4, 8, 3); // Resultado incorrecto
```

Observe que el parámetro `el_menor_entre_ambos` está completamente determinado por 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 401, en la que se calcula el menor dentro de la función.

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/jccubero/FP/IV_mcd_funcion.cpp

Los ejemplos anteriores ilustran las siguientes normas de diseño de funciones:

Todos los datos auxiliares que la función necesite para realizar sus cálculos serán datos locales.

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.

IV.1.2.3. ¿Qué parámetros pasamos?

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 cuáles eran 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 416.

```
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/jccubero/FP/IV_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 155, 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 160, 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 << "Mayoría de edad y altura.\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 y así determinar los parámetros apropiados.

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 205:

```
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/jccubero/FP/IV_altura.cpp

Nota:

Si las constantes se utilizasen en otros sitios del programa, podrían ponerse como constantes globales

IV.1.3. Cuestiones varias

IV.1.3.1. Ámbito de un dato (revisión). Variables globales

Recuerde que el **ámbito (scope)** 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 (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 pueden producir efectos laterales

IV.1.3.2. 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 se produce un comportamiento indeterminado (página 318)

Ejemplo. Considere 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, evite la inclusión de sentencias `return` perdidas dentro del código de la función. Use mejor un único `return` al final de la función.*



IV.1.3.3. Sobrecarga de funciones (Ampliación)

C++ permite definir *sobrecarga de funciones (function overload)*, es decir, varias funciones con el mismo nombre pero que difieren:



- ▷ O bien en el número de parámetros
- ▷ O bien en el tipo de los argumentos

El compilador llamará a la versión que mejor encaje.

Dos versiones sobrecargadas no pueden diferenciarse, únicamente, en el tipo de dato devuelto, ya que el compilador no sabría a cuál llamar, ya que la expresión resultante podría asignarse a datos de diferentes tipos (debido al casting automático)

Ejemplo.

```
double Max(double uno, double otro){           // Versión 1
    .....
}
double Max(double uno, double otro, double tercero){
    .....
}
/*
// Error de compilación:
float Max(double uno, double otro){           // Versión 2
    .....
}
*/
int main(){
    .....
    max = Max(3.5, 4.3);           // Versión 1
    max = Max(3.5, 4.3, 8.5);     // Versión 2
    .....
}
```

Si el compilador encuentra una llamada **ambigua**, dará un error en tiempo de compilación:

Ejemplo.

```
double Max(double uno, double otro){           // Versión 1
    .....
}
long Max(long uno, long otro){                 // Versión 2
    .....
}

int main(){
    int a, b, max;
    .....
    max = Max(a, b);           // Error de compilación
    .....
}
```

Los parámetros actuales `a` y `b` pueden ser transformados tanto a un `double` como a un `long`, por lo que la llamada es ambigua y el programa no se puede compilar.

Consejo: *No abuse de la sobrecarga de funciones que únicamente difieran en los tipos de los parámetros formales*



IV.2. Clases

Hemos visto cómo las funciones introducen mecanismos para cumplir dos principios básicos:

- ▷ Principio de una 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 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.

IV.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 es un dato compuesto que 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(), Agranda(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:** ConstruyeHipotenusa(), ConstruyeSegmentoAB(), etc.

Objeto: una fracción

- ▷ **Datos:** Numerador y denominador
- ▷ **Métodos:** Simplifica(), Súmale(Fracción otra_fracción), etc.

Objeto: una calculadora de nómina

- ▷ **Datos:** salario base
- ▷ **Métodos:** AplicaSubidaSalarial(int edad, int num_hijos)

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. 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. como cualquier otro dato, para declarar un objeto habrá que indicar primero el tipo de dato (el nombre de la clase) y a continuación el nombre del dato (el del objeto).

```
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`.

Una clase es un tipo de dato.

Los objetos son datos cuyo tipo de dato es una clase.

IV.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 el mecanismo de modularización utilizado en PDO. La idea consiste en aunar datos y comportamiento en un mismo módulo.

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)* :

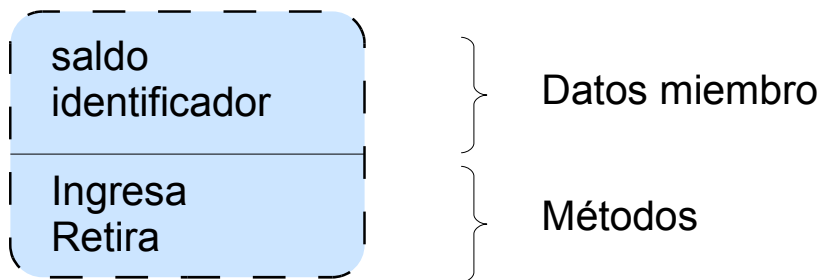
Son funciones definidas dentro de la clase.

Determinan el *comportamiento (behaviour)* de la entidad, es decir, el conjunto de operaciones que se pueden realizar sobre los objetos de la clase.

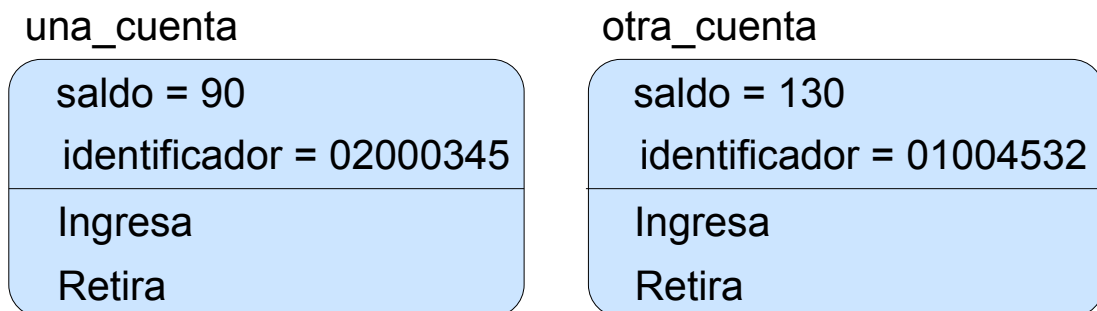
La definición de los métodos es la misma para todos los objetos.

```
class CuentaBancaria{                // <- clase
    .....
};
int main(){
    CuentaBancaria una_cuenta;        // <- un objeto
    CuentaBancaria otra_cuenta;       // <- otro objeto
    .....
}
```

Clase CuentaBancaria:



Objetos instancia de CuentaBancaria:



IV.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.

Cuando se crea un objeto, los datos miembro, como cualquier otro dato, contendrán un valor indeterminado.

```
class MiClase{
public:
    int dato;
};

int main(){
    MiClase un_objeto;      // un_objeto.dato   contiene   ?
    MiClase otro_objeto;    // otro_objeto.dato contiene   ?

    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;                   // ?
    double 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 sobre la lectura:

▷ 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
```

Cuando se crea un objeto, los datos miembro no tienen ningún valor asignado por defecto. Por lo tanto, después de la sentencia

```
CuentaBancaria cuenta;
```

el valor del dato miembro `saldo` es indeterminado (representado por ?) Sin embargo, el valor del dato miembro `identificador` sí tiene un valor específico: la cadena vacía. Recuerde (ver página 84) que C++ inicializa a cadena vacía todos los datos de tipo `string`. Lo mismo ocurre con la variable `identificador_cuenta` del programa principal.

Si deseamos dar un valor inicial a los datos miembro de cualquier objeto de una clase, se puede especificar en la definición de ésta. En este caso, al crear el objeto, se le asignarán automáticamente los valores especificados en la inicialización. Esto es posible a partir de C++11.

```
class CuentaBancaria{  
public:  
    double saldo          = 0.0;  
    string identificador = "";  
};
```



```
int main(){  
    CuentaBancaria cuenta;           // 0, ""  
    CuentaBancaria otra_cuenta;      // 0, ""  
  
    cout << cuenta.saldo << "\n";    // 0  
    cout << cuenta.identificador;    // ""
```

Observe que la inicialización del dato miembro `identificador`

```
string identificador = "";
```

no es necesaria, en el sentido que hemos comentado anteriormente, ya que C++ inicializa todos los datos de tipo `string` a la cadena vacía. En cualquier caso, a lo largo de la asignatura lo haremos explícitamente como buen hábito de programación, extensible a cualquier otro lenguaje.

Las inicializaciones especificadas en la declaración de los datos miembro dentro de la clase son posibles desde C++11. Éstas se aplican en el momento que se crea un objeto cualquiera de dicha clase.

IV.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.

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. Empezamos viendo los 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 básicamente el de Google (Buscar en Internet [Google coding style](#)):**
 - **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`
- ▷ **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 = 0.0;
    string identificador = "";

    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"; // 0, "2...9"
    una_cuenta.Ingresa(25); // 25, "2...9"
    una_cuenta.Retira(10); // 15, "2...9"
    .....
    otra_cuenta.identificador = "20310381450100007518"; // 0, "2...8"
    otra_cuenta.Ingresa(45); // 45, "2...8"
    otra_cuenta.Retira(15); // 30, "2...8"
    .....
}
```



Nota:

Observe que en la llamada `una_cuenta.Ingresar(25)` estamos pasando como parámetro un `int` a un `double`. Al igual que ocurría con las funciones, el paso de parámetros sigue las mismas directrices por lo que se producirá un casting automático y la correspondencia entre el parámetro actual y el formal se hará correctamente.

A destacar:

- ▷ **Los métodos `Ingresar` y `Retira` acceden al dato miembro `saldo` por su nombre. En general, podemos decir que:**

Los métodos acceden a los datos miembro directamente.

- ▷ **Los métodos `Ingresar` y `Retira` modifican el dato miembro `saldo`.**

Los métodos pueden modificar el estado del objeto.

- ▷ **Aplicaremos la siguiente norma a la hora de elegir un identificador para los métodos. Esta norma es aplicable en cualquier lenguaje de programación, no únicamente C++.**

Usaremos nombres para denotar las clases y verbos para los métodos de tipo `void`. Normalmente no usaremos infinitivos sino la tercera persona del singular.

Para los métodos que devuelven un valor, usaremos nombres (el del valor que devuelve).

Ejemplo. Definamos la clase `SegmentoDirigido` para representar el segmento delimitado por dos puntos del plano.

```
class SegmentoDirigido{
    public:
        double x_1, y_1, x_2, y_2;
};

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 << un_segmento.x_1;    // 3.4
    cout << un_segmento.x_2;    // 4.5
}
```



Recuerde que si lo considera oportuno, se puede especificar la inicialización de 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,        // 0.0, 0.0, 0.0, 0.0
                     otro_segmento;      // 0.0, 0.0, 0.0, 0.0

    cout << un_segmento.x_1 << " " << otro_segmento.x_1; // 0.0 0.0
}
```



Ejemplo. Defina sendos métodos `TrasladaHorizontal` y `TrasladaVertical` para trasladar un segmento un número de unidades.



```
class SegmentoDirigido{
public:
    double x_1,
           y_1,
           x_2,
           y_2;

    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 << "," << un_segmento.y_1;    // 13.4,5.6
    cout << un_segmento.x_2 << "," << un_segmento.y_2;    // 14.5,2.3
}
```

Ejercicio. La longitud de un segmento dirigido delimitado por los puntos (x_1, y_1) y (x_2, y_2) viene dada por la fórmula:

$$l = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Añada un método a la clase `SegmentoDirigido` para calcular su longitud.

IV.2.2.3. Llamadas entre métodos dentro del propio objeto

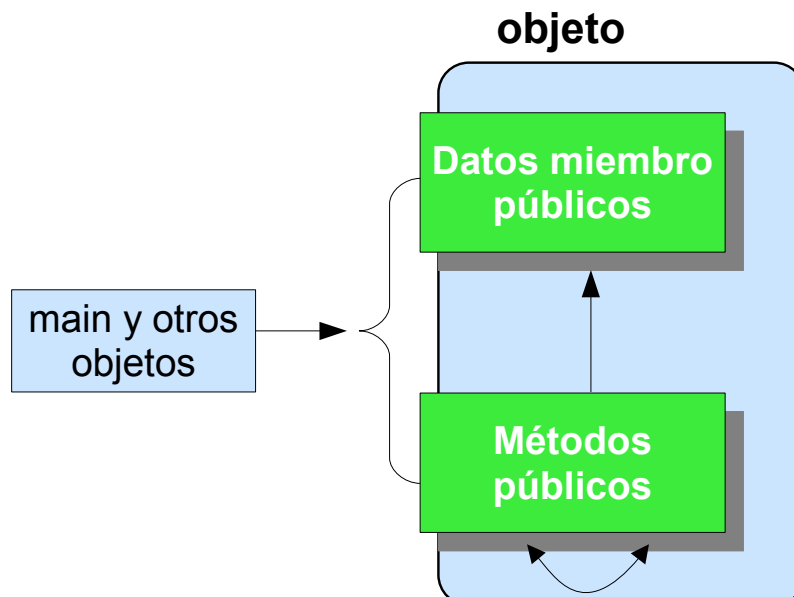
Ya sabemos cómo ejecutar los métodos sobre un objeto:

```
objeto.UnMetodo(...);
```

¿Pero pueden llamarse dentro del objeto unos métodos a otros? Si.

Dentro de una clase, todos los métodos pueden llamar a otros métodos de la misma clase. 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. Por lo tanto, la llamada se realiza como si fuese una función.

```
class MiClase{  
public:  
    void UnMetodo(parámetros formales){  
    }  
    void OtroMetodo(...){  
        UnMetodo(parámetros actuales);  
    }  
};
```



Ejemplo. Implemente el método `Traslada` para que traslade el segmento tanto en horizontal como en vertical.

```
class SegmentoDirigido{
public:
    double x_1, y_1,
           x_2, y_2;
    void TrasladaHorizontal(double unidades){ ..... }
    void TrasladaVertical(double unidades){ ..... }

    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.Traslada(5.0, 10.0);
    .....
}
```

http://decsai.ugr.es/jccubero/FP/IV_segmento_dirigido_todo_public.cpp

Ejemplo. Aplique un interés porcentual al saldo de la cuenta bancaria.

```
class CuentaBancaria{
public:
    double saldo          = 0.0;
    string identificador;

    void AplicaInteresPorcentualRepitiendoCodigo(int tanto_porcentaje){
        double cantidad;
        cantidad = saldo * tanto_porcentaje / 100.0;

        saldo = saldo + cantidad;
    }

    void AplicaInteresPorcentual (int tanto_porcentaje){
        Ingresa (saldo * tanto_porcentaje / 100.0);
    }

    void Ingresa(double cantidad){
        saldo = saldo + cantidad;
    }

    void Retira(double cantidad){
        saldo = saldo - cantidad;
    }
};

int main(){
    CuentaBancaria cuenta; // 0, ""

    cuenta.identificador = "20310381450100006529"; // 0, "2...9"
    cuenta.Ingresa(25); // 25, "2...9"
    cuenta.AplicaInteresPorcentual(3); // 25.75, "2...9"
```



488



http://decsai.ugr.es/jccubero/FP/IV_cuenta_bancaria_todo_public.cpp

En resumen:

Dentro de la clase, los métodos pueden llamarse unos a otros. Esto nos permite cumplir el principio de una única vez.

IV.2.2.4. 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 431) 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          = 0.0;
    string identificador = "";
    // Realmente, C++ ya 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";    // 0, "2...9"

    cin >> ingreso;

    if (ingreso > 0)
        cuenta.Ingresa(ingreso);

    cin >> retirada;

    if (retirada > 0 && retirada <= cuenta.saldo)
        cuenta.Retira(retirada);
```

Si lo hacemos de esa forma, **cada vez** que realicemos un ingreso o retirada de fondos, habría que realizar las anteriores comprobaciones, por lo que es propenso a errores, ya que seguramente, alguna vez no lo haremos.

Solución: Lo programamos dentro del método correspondiente. Siempre que se ejecute el método se realizará la comprobación automáticamente.



```
class CuentaBancaria{
public:
    double saldo          = 0.0;
    string identificador = "";

    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"; // 0, "2...9"
    cuenta.Ingresa(-3000); // 0, "2...9"
    cuenta.Ingresa(25);    // 25, "2...9"
    cuenta.Retira(-10);    // 25, "2...9"
    cuenta.Retira(10);     // 15, "2...9"
    cuenta.Retira(100);    // 15, "2...9"
```

En resumen:

Los métodos permiten establecer la política de acceso a los datos miembro, es decir, determinar cuáles son las operaciones permitidas con ellos.

IV.2.3. Ocultación de información

IV.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, siempre deberíamos usar el método `Ingresar` y no acceder directamente al dato miembro `cuenta.saldo`. ¿Cómo lo imponemos? 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)* .

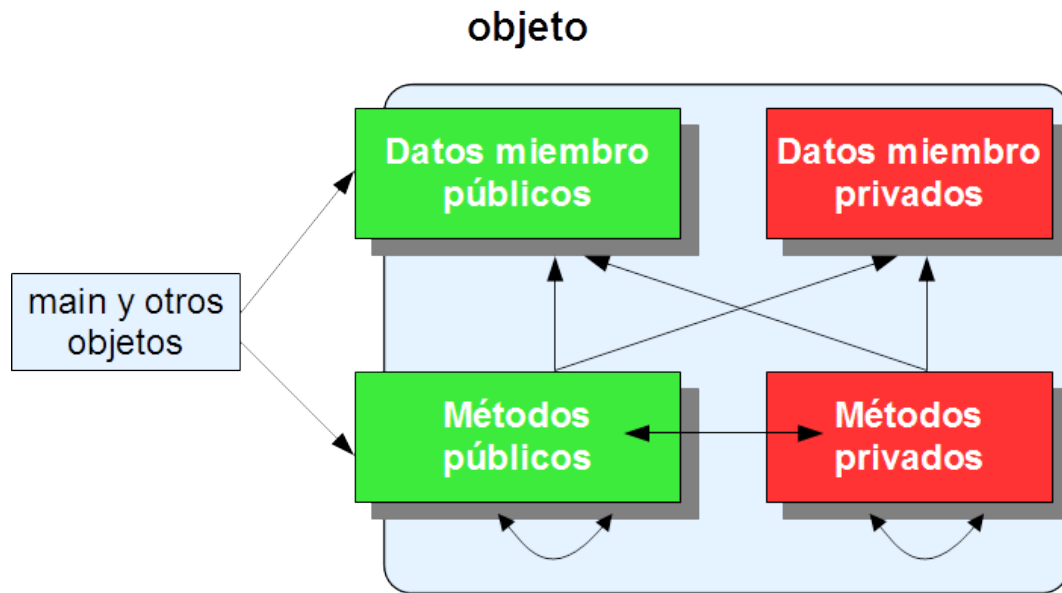
- Se indica con el especificador de ámbito `private` (éste es el ámbito por defecto si no se pone nada)

```
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.



IV.2.3.2. UML: Unified modeling language

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 datos miembro y métodos públicos se notarán con un símbolo + y los privados con -

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*

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.

IV.2.3.3. Métodos Get y Set: protegiendo los datos miembro

Supongamos que cambiamos el ámbito público de los datos miembro de la clase Cuenta_Bancaria a privado.

```
class CuentaBancaria{
private:
    double saldo          = 0.0;
    string identificador = "";
    // Realmente, C++ ya 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_porcentaje){
        Ingresa (saldo * tanto_porcentaje / 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;
```

Entonces, ¿cómo consultamos o asignamos un valor a los datos miembro?

Al trabajar con datos miembro privados debemos añadirle a la clase:

- ▷ **Métodos Set** : Métodos públicos para modificar los datos miembro. Normalmente serán métodos `void` que recibirán como parámetro un(os) valor(res) y el método lo(s) asignará(n) al correspondiente dato miembro.

Normalmente, los nombraremos añadiendo al nombre el prefijo `Set`.

- ▷ **Métodos Get** : Métodos para obtener el valor actual de los datos miembro.

Normalmente, los nombraremos igual que el valor que devuelven. En casos excepcionales añadiremos al nombre el prefijo `Get`.

Esto nos obliga a escribir más líneas de código, pero es conveniente que lo hagamos, ya que así controlamos qué datos pueden asignarse a los datos miembro o, por ejemplo, si se asignan todos de golpe o uno a uno.

Salvo casos excepcionales (ver página 565), no definiremos datos miembro públicos. Siempre serán privados.

Esto nos permitirá controlar, desde dentro de la clase, las operaciones que puedan realizarse sobre ellos. Estas operaciones estarán encapsuladas en métodos de la clase.

Desde fuera de la clase, no se tendrá acceso directo a los datos miembro privados. El acceso será a través de los métodos anteriores.



Ejemplo. Cambiemos el ámbito (de `public` a `private`) de los datos miembro de la clase `SegmentoDirigido`. Podemos elegir entre:

- ▷ Cambiar cada coordenada por separado.
- ▷ Cambiar las cuatro a la vez.

Cambiando cada una por separado:

```
class SegmentoDirigido{
private:
    double x_1, y_1, x_2, y_2;
public:
    void SetOrigenAbscisa(double origen_abscisa){
        x_1 = origen_abscisa;
    }
    void SetOrigenOrdenada(double origen_ordenada){
        y_1 = origen_ordenada;
    }
    void SetFinalAbscisa(double final_abscisa){
        x_2 = final_abscisa;
    }
    void SetFinalOrdenada(double final_ordenada){
        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;           // ?, ?, ?, ?

    un_segmento.SetOrigenAbscisa(3.4);       // 3.4, ?, ?, ?
    un_segmento.SetOrigenOrdenada(5.6);     // 3.4, 5.6, ?, ?
    un_segmento.SetFinalAbscisa(4.5);       // 3.4, 5.6, 4.5, ?
    un_segmento.SetFinalOrdenada(2.3);      // 3.4, 5.6, 4.5, 2.3

    cout << "Segmento Dirigido.\n\n";
    cout << "Antes de la traslación:\n";
    cout << un_segmento.OrigenAbscisa() << " , "
        << un_segmento.OrigenOrdenada();    // 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.OrigenAbscisa() << " , "
        << un_segmento.OrigenOrdenada();    // 8.4 , 15.6
    cout << "\n";
    cout << un_segmento.FinalAbscisa() << " , "
        << un_segmento.FinalOrdenada();    // 9.5 , 12.3
```

Cambiando 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, y_1, x_2, y_2;
public:
    void SetCoordenadas(double origen_abscisa, double origen_ordenada,
                        double final_abscisa, double final_ordenada){
        x_1 = origen_abscisa;
        y_1 = origen_ordenada;
        x_2 = final_abscisa;
        y_2 = final_ordenada;
    }
    .....
    // El resto de métodos no varían
};

int main(){
    SegmentoDirigido un_segmento;  // ?, ?, ?, ?

    un_segmento.SetCoordenadas(3.4, 5.6, 4.5, 2.3);  // 3.4, 5.6, 4.5, 2.3

    // El resto del main sigue igual
```

Ejercicio. Cree una clase que represente una circunferencia. Vendrá determinada por las dos coordenadas del centro y por la longitud del radio, que serán sus datos miembro. Proporcione los correspondientes métodos `Get` y decida qué método(s) utilizará para cambiar sus valores.

http://decsai.ugr.es/jccubero/FP/IV_circunferencia.cpp

Ejemplo. Defina 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	Dia()
+ int	Mes()
+ int	Anio()
+ string	ToString()

```
class Fecha{
private:
    int dia,
        mes,
        anio;
public:
    void SetDiaMesAnio(int el_dia, int el_mes, int el_anio){
        dia = el_dia;
        mes = el_mes;
        anio = el_anio;
    }
    int Dia(){
        return dia;
    }
    int Mes(){
        return mes;
    }
    int Anio(){
        return anio;
    }
    string ToString(){
```



553

```
        return to_string(dia) + "/" +    // to_string standard en C++11
               to_string(mes) + "/" +
               to_string(anio);

        // También sería correcto llamar a los métodos
        // Dia(), Mes(), Anio()
    }
};

int main(){
    Fecha nacimiento_JC;                // ?, ?, ?

    nacimiento_JC.SetDiaMesAnio(27, 2, 1967); // 27, 2, 1967

    cout << nacimiento_JC.ToString();      // Imprime 27/2/1967
}
```

Ejemplo. Cuenta bancaria. Añadimos métodos para modificar y obtener el valor del saldo y del identificador.

```
class CuentaBancaria{  
private:  
    double saldo          = 0.0;  
    string identificador = "";  
    // Realmente, C++ ya inicializa los string a ""  
public:  
    void SetIdentificador(string identificador_cuenta){  
        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_porcentaje){  
        Ingresa (saldo * tanto_porcentaje / 100.0);  
    }  
}
```



```
};
int main(){
    CuentaBancaria cuenta;    // "", 0

    cuenta.SetIdentificador("20310381450100006529");    // "2...9", 0
    cuenta.Ingresar(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 Ingresar y Retira. Para ello, basta eliminar SetSaldo.

http://decsai.ugr.es/jccubero/FP/IV_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	Ingresar(double cantidad)
+ void	Retira(double cantidad)
+ void	AplicaInteresPorcentual(int tanto_por_ciento)

IV.2.3.4. Datos miembro vs parámetros de los métodos

A la hora de diseñar una clase tenemos que decidir qué incluimos como datos miembro. En principio, lo normal es que tengamos pocos datos miembro. Así pues, si un método necesita usar un dato y éste no se va a usar en otros métodos, lo declararemos local al método. Y si el método necesita información del exterior, se le pasará como parámetro. Reservaremos los datos miembro a aquellos datos que son utilizados por muchos métodos, lo que nos indica que, de alguna forma, definen la *esencia* de un objeto de la clase.

Por ejemplo, si un dato d se puede obtener a partir de otros datos miembro, d no será un dato miembro sino que definiremos un método para obtenerlo:

Ejemplo. Sobre la clase `SegmentoDirigido`, ¿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.

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!

Ejemplo. ¿Cómo se definiría la longitud de un segmento con una función y no dentro de la clase `SegmentoDirigido`?

▷ Con una función debemos pasar 4 parámetros:

```
Longitud(double x_1, double y_1, double x_2, double y_2){
    .....
}
int main(){
    .....
    longitud_segmento = Longitud(1.0, 2.0, 3.0, 4.0);
```

▷ Dentro de la clase, no:

SegmentoDirigido
- double x_1
- double y_1
- double x_2
- double y_2
.....
+ double Longitud()

```
int main(){
    SegmentoDirigido segmento;
    .....
    // Asignación de valores con los métodos Set -> 1.0, 2.0, 3.0, 4.0
    longitud_segmento = segmento.Longitud();
```

Ejemplo. Defina la clase `Persona`. Representaremos su nombre, altura y edad.

Persona	
- string	nombre
- int	edad
- int	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;
public:
    string Nombre(){
        return nombre;
    }
    int Edad(){
        return edad;
    }
    int Altura(){
        return altura;
    }
    void SetNombre(string nombre_persona){
        nombre = nombre_persona;
    }
    void SetEdad(int edad_persona){
        edad = edad_persona;
    }
}
```

```
void SetAltura(int altura_persona){  
    altura = altura_persona;  
}  
};
```

En la página 436 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 dentro de la clase? Bastaría añadir los siguientes métodos

```
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();  
}
```

Observe la diferencia:

▷ Con funciones:

```
.....  
es_mayor_edad = EsMayorEdad(48);  
es_alta       = EsAlta(186, 48);
```

▷ Con una clase:

```
Persona una_persona("JC", 48, 186);  
.....  
es_mayor_edad = una_persona.EsMayorEdad();  
es_alta       = una_persona.EsAlta();
```

En el segundo caso, los datos importantes de la persona, ya están dentro del objeto `una_persona` y no hay que pasarlos como parámetros una y otra vez.

http://decsai.ugr.es/jccubero/FP/IV_persona.cpp

Nota:

La edad es un dato temporal, por lo que sería mucho mejor representar la fecha de nacimiento. Lo podremos hacer en el tema V cuando veamos cómo declarar un objeto como dato miembro de otro.

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.

```
SegmentoDirigido un_segmento (1.0, 2.0, 1.0, 3.0);  
un_segmento.TrasladaHorizontal(4);
```

Fomentaremos el uso de datos locales en los métodos de las clases. Sólo incluiremos como datos miembro aquellas características esenciales que determinan una entidad.

Si una propiedad de la clase se puede calcular a partir de los datos miembro, no será a su vez un dato miembro sino que su valor se obtendrá a través de un método de la clase.

La información adicional que se necesite para ejecutar un método, se proporcionará a través de los parámetros de dicho método.



En la página 622 se ampliará este apartado en aquellos casos en los que no está claro si un dato ha de ser dato miembro o no.

IV.2.3.5. Comprobación de precondiciones en los métodos `Set`

Normalmente habrá una restricción sobre los valores a asignar a los datos miembro:

- ▷ En un `SegmentoDirigido` no permitimos segmentos degenerados, es decir, con el mismo origen y final
- ▷ En una `Circunferencia` no permitimos radios negativos.
- ▷ En una `Fecha` los días deben estar entre 1 y 31 y además depende del mes (que tiene que estar entre 1 y 12)
- ▷ En una `CuentaBancaria` el identificador ha de tener 21 dígitos y debe ser correcto el valor de control (dos dígitos dentro de la secuencia)

¿Dónde realizamos la comprobación?

- ▷ ¿En cada programa principal que construyamos, comprobando que los parámetros actuales sean correctos?
- ▷ ¿Dentro del correspondiente método `Set` comprobando que los parámetros formales sean correctos?

Lo normal será que lo hagamos dentro del correspondiente método `Set`, aunque habrá ocasiones en la que esto no sea así. Cada opción tiene sus ventajas e inconvenientes.

Ejemplo. Supongamos que queremos impedir que un segmento dirigido sea *degenerado*, es decir, que tenga el mismo punto inicial y final. Si hacemos la comprobación dentro de cada `main`:

```
// un main
int main(){
    SegmentoDirigido segmento;
    .....
    cin >> x1 >> x2 >> y1 >> y2;

    if (! (x1 == x2 && y1 == y2) )
        segmento.SetCoordenadas(x1, y1, x2, y2);
    else
        .....
```

```
// otro main
int main(){
    SegmentoDirigido sgm;
    .....
    cin >> abs1 >> abs2 >> ord1 >> ord2;

    if (! (abs1 == abs2 && ord1 == ord2) )
        sgm.SetCoordenadas(abs1, ord1, abs2, ord2);
    else
        .....
```

Vemos que las sentencias condicionales de los dos programas son iguales (salvo los nombres de las variables) Esta es una forma de duplicar código entre programas distintos, violando por tanto el principio de una única vez (página 112). Será muy posible que en algún `main` se nos olvide realizar correctamente la comprobación.

Hagamos la comprobación dentro del método:

```
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;
    }
}
```

```
// un main
int main(){
    SegmentoDirigido segmento;
    .....
    cin >> x1 >> x2 >> y1 >> y2;
    segmento.SetCoordenadas(x1, y1, x2, y2);
```

```
// otro main
int main(){
    SegmentoDirigido sgm;
    .....
    cin >> abs1 >> abs2 >> ord1 >> ord2;
    sgm.SetCoordenadas(abs1, ord1, abs2, ord2);
```

Ahora, la clase es la encargada de comprobar que los valores son correctos. 😊

Ahora bien, realizar la comprobación en cada `main` tiene una ventaja: la acción a realizar en el `else` se puede particularizar para cada `main`. Si la hacemos dentro de la clase, la acción será siempre la misma. Sobre esto volveremos en la página 538. Por ahora, aceptamos este inconveniente como mal menor.

¿Qué acción suele ejecutarse en el `else` del condicional de un método `Set` cuando éste comprueba si los parámetros actuales son correctos? Normalmente no haremos nada.

```
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;
    }
    // else <-- No hay else
}

int main(){
    SegmentoDirigido segmento;

    segmento.SetCoordenadas(1.0, 2.0, 3.0, 4.0); // 1.0, 2.0, 3.0, 4.0
    segmento.SetCoordenadas(1.0, 2.0, 1.0, 2.0); // 1.0, 2.0, 3.0, 4.0
    .....
}
```

La sentencia `segmento.SetCoordenadas(1.0, 2.0, 1.0, 2.0);` no modifica las coordenadas ya que la expresión condicional del método es `false` (es un segmento degenerado) y no se realiza la asignación.

En cualquier caso, también es un diseño correcto no realizar ninguna comprobación dentro del método `Set` e imponer como precondition de uso de dicho método que los parámetros pasados sean correctos (ver más detalle en la página 538)

En resumen:

Será usual que, dentro de los métodos `Set`, comprobemos que los parámetros pasados sean correctos. Si no lo son, lo normal será que no hagamos nada dentro del método y mantengamos los que hubiese.

En cualquier caso, también es lícito no realizar ninguna comprobación e imponer la correspondiente restricción como precondition del método `Set`.

Ejemplo. Comprobamos que la fecha es correcta en el correspondiente método Set.

Fecha	
- int	dia
- int	mes
- int	anio
+ void	SetDiaMesAnio (int el_dia, int el_mes, int el_anio)
+ int	Dia()
+ int	Mes()
+ int	Anio()
+ string	ToString()

```
class Fecha{
private:
    int dia,
        mes,
        anio;
public:
    void SetDiaMesAnio(int el_dia, int el_mes, int el_anio){
        bool es_bisiesto;
        bool es_fecha_correcta;
        const int anio_inferior = 1900;
        const int anio_superior = 2500;
        const int dias_por_mes[12] =
            {31,28,31,30,31,30,31,31,30,31,30,31};
            // Meses de Enero a Diciembre

        es_fecha_correcta = 1 <= el_dia &&
                            el_dia <= dias_por_mes[el_mes - 1] &&
                            1 <= el_mes && el_mes <= 12 &&
                            anio_inferior <= el_anio &&
                            el_anio <= anio_superior;

        es_bisiesto = (el_anio % 4 == 0 && el_anio % 100 != 0) ||
```



```
        el_anio % 400 == 0;

    if (!es_fecha_correcta && el_mes == 2 && el_dia == 29 && es_bisiesto)
        es_fecha_correcta = true;

    if (es_fecha_correcta){
        dia = el_dia;
        mes = el_mes;
        anio = el_anio;
    }
}

int Dia(){
    return dia;
}

int Mes(){
    return mes;
}

int Anio(){
    return anio;
}

string ToString(){
    return to_string(dia) + "/" +    // to_string standard en C++11
           to_string(mes) + "/" +
           to_string(anio);

    // También sería correcto llamar a los métodos
    // Dia(), Mes(), Anio()
}

};

int main(){
    Fecha nacimiento_JC;                // ?, ?, ?
    Fecha otra_fecha;                    // ?, ?, ?
```

```
nacimiento_JC.SetDiaMesAnio(27, 2, 1967); // 27, 2, 1967
nacimiento_JC.SetDiaMesAnio(-8, 2, 1967); // 27, 2, 1967

cout << nacimiento_JC.ToString();           // Imprime 27/2/1967

otra_fecha.SetDiaMesAnio(8, -2, 1967);      // ?, ?, ?
```

Nota:

Normalmente, no tendremos variables como `nacimiento_JC` ligadas a una persona en concreto. Lo usual es que tengamos un conjunto de objetos de tipo **Fecha**, asociados a distintas personas según un índice y guardados en un vector de objetos. Esto se verá en el próximo tema.

Nota:

Observe que se ha definido un vector como dato local de un método. Esto es correcto. El vector vivirá mientras se esté ejecutando el método. Consulte la página [568](#) para más detalle.

IV.2.3.6. 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 `Ingresa` 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 Ingresa(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);
    }
}
```

Observe que en el caso de que el saldo propuesto sea incorrecto, se deja el valor antiguo.

Recordemos que SetSaldo no queríamos que fuese `public` ya que sólo permitíamos ingresos y retiradas de fondos y no asignaciones directas.

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/jccubero/FP/IV_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. Reescriba el ejemplo del segmento de la página 496 para que la comprobación de que las coordenadas son correctas se hagan en un método privado.

```
class SegmentoDirigido{
private:
    double x_1, y_1, x_2, y_2;

    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/jccubero/FP/IV_segmento_dirigido_metodos_privados.cpp

Ejemplo. Reescriba el ejemplo de la Fecha de la página 485 para que la comprobación de que la fecha es correcta se haga en un método privado.

Fecha	
- int	dia
- int	mes
- int	anio
- bool	EsFechaCorrecta (int el_dia, int el_mes, int el_anio)
+ void	SetDiaMesAnio(int el_dia, int el_mes, int el_anio)
+ int	Dia()
+ int	Mes()
+ int	Anio()
+ string	ToString()

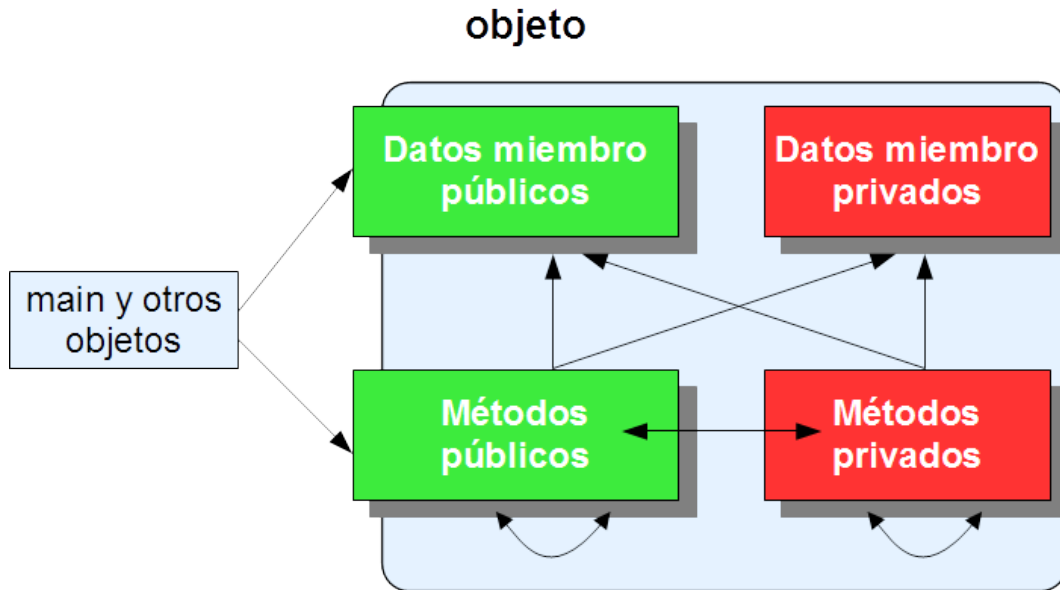
```
class Fecha{
private:
    int dia,
        mes,
        anio;

    bool EsFechaCorrecta(int el_dia, int el_mes, int el_anio){
        .....
    }
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;
        }
    }
    .....
};
```

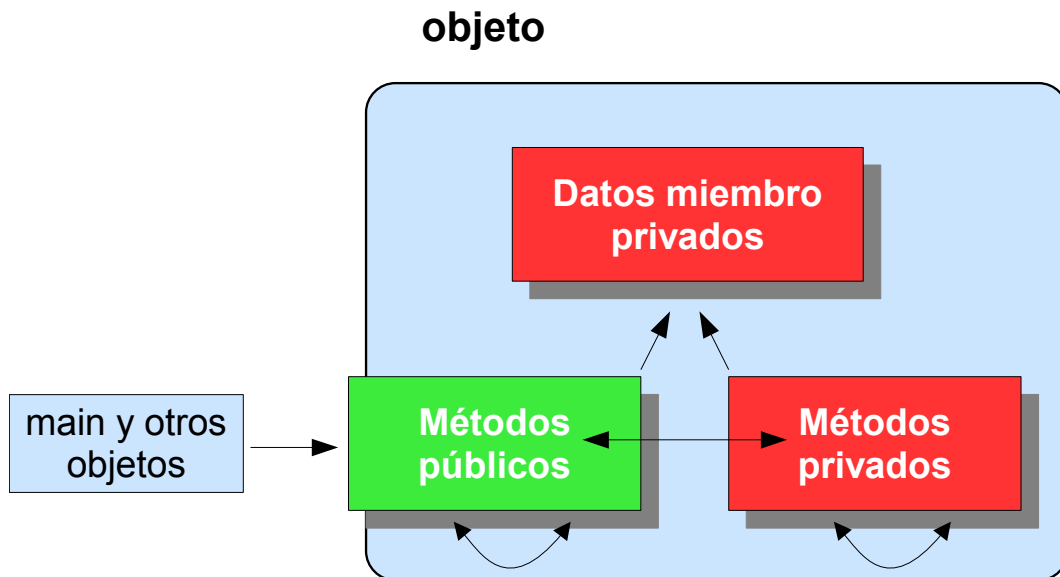


553

En resumen. Lo que C++ permite:



Lo que nosotros haremos:



IV.2.3.7. Ámbito de un objeto

En los temas I y II (páginas 39 y 175) vimos el concepto de ámbito de una variable de un tipo simple. Lo mismo se aplica cuando la variable es un objeto instancia de una clase.

Por ejemplo, podemos declarar un objeto dentro de un bucle. Cada vez que se produzca una iteración, se volverá a crear la variable. Para ello, el compilador debe liberar la memoria asociada a la variable en la iteración anterior y volver a reservar memoria en la iteración actual. Esto puede suponer una recarga computacional que habrá que tener en cuenta.

- ▷ **Versión 1. Se crea un objeto nuevo en cada iteración:**

```
while (hay_datos){
    cin >> dia >> mes >> anio;
    hay_datos = (anio != -1);

    if (hay_datos){
        Fecha fecha_nacimiento;
        fecha_nacimiento.SetDiaMesAnio(dia, mes, anio);
        cout << fecha_nacimiento.ToString();
    }
}
```

- ▷ **Versión 2. Sólo se crea un objeto fuera del bucle y se van modificando sus datos miembro en cada iteración.**

```
Fecha fecha_nacimiento;

while (hay_datos){
    cin >> dia >> mes >> anio;
    hay_datos = (anio != -1);

    if (hay_datos){
```

```
        fecha_nacimiento.SetDiaMesAnio(dia, mes, anio);  
        cout << fecha_nacimiento.ToString();  
    }  
}
```

Lo normal es que prefiramos la segunda versión, ya que no requiere la creación de múltiples objetos. Volveremos a este problema en la página **536**

IV.2.4. Constructores

IV.2.4.1. Definición de constructores

C++ utiliza un **constructor de oficio** sin parámetros, oculto, que permite crear el objeto y poco más.

```
class MiClase{
    .....
};
int main(){
    MiClase objeto; // Constructor de oficio de C++
```

¿Y si queremos realizar ciertas acciones iniciales en el momento de la definición del objeto? Debemos definir 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.
- ▷ Cada vez que se cree un objeto, el compilador ejecutará las instrucciones especificadas en el constructor.
- ▷ Se puede incluir parámetros, en cuyo caso, habrá que incluir los correspondientes parámetros actuales en el momento de la definición de cada objeto. Si no se incluyen, se produce un error de compilación.

Para asignar dichos valores a los datos miembro, se puede hacer con el operador de asignación, pero es más recomendable a través de la **lista de inicialización del constructor (constructor initialization list)**.

Son construcciones del tipo `<dato_miembro> (<valor inicial>)` separadas por coma. La lista de inicialización del constructor va antes del paréntesis del constructor, y con dos puntos al inicio.

- ▷ **Si el programador define cualquier constructor (con o sin parámetros) ya no está disponible el constructor de oficio.**

Ejemplo. Añada un constructor a la cuenta bancaria pasándole como parámetro el identificador de cuenta.

```
class CuentaBancaria{
private:
    double saldo          = 0.0;
    string identificador;
    .....
public:
    /*
    CuentaBancaria(string identificador_cuenta){ // Correcto
        identificador = identificador_cuenta;
    }
    */
    CuentaBancaria(string identificador_cuenta) // Preferible
        :identificador(identificador_cuenta) // <- Lista inicialización
    {
    }
    .....
};

int main(){
    // CuentaBancaria cuenta; <- Error de compilación

    CuentaBancaria cuenta("20310381450100007510");
    // cuenta: "20310381450100007510", 0.0
```

Si los parámetros actuales son variables, habrá que crear el objeto después de establecer dichas variables.

```
cin >> identificador;

CuentaBancaria cuenta(identificador);
```

Ejemplo. Añadimos un constructor a la clase `Fecha` que definimos en la página 508 para obligar a crear el objeto con los datos del día, mes y año.

Fecha	
- int	dia
- int	mes
- int	anio
+	Fecha (int el_dia, int el_mes, int el_anio)
+

```
class Fecha{
private:
    int dia,
        mes,
        anio;
    .....
public:
    Fecha(int el_dia, int el_mes, int 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
```



Ejercicio. Añada un constructor a la clase `SegmentoDirigido` para obligar a que, en el momento de la definición de cualquier objeto, se pasen como parámetros las cuatro coordenadas del segmento.

En un constructor podemos ejecutar las instrucciones que deseamos que se ejecuten en el momento de la creación de cualquier objeto de la clase.

Ejemplo. Dentro del constructor de la cuenta bancaria llamamos a un método (en este caso privado) para obtener el código de seguridad de un identificador (el suministrado como parámetro en el constructor):

```
class CuentaBancaria{
private:
    double saldo          = 0.0;
    string identificador_IBAN;
    string GetCodigoControl(string id_CCC){
        .....
    }
public:
    CuentaBancaria(string identificador_CCC){
        string codigo_control;
        codigo_control = GetCodigoControl(identificador_CCC);
        identificador_IBAN = "IBAN ES" +
                               codigo_control +
                               " " +
                               identificador_CCC;
    }
    .....
};

int main(){
    CuentaBancaria cuenta("20310381450100007510");
    // cuenta: "IBAN ES 17 20310381450100007510", 0.0
```

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)
+	CuentaBancaria(string identificador_cuenta)
+ string	Identificador()
+ double	Saldo()
+ void	Ingresa(double cantidad)
+ void	Retira(double cantidad)
+ void	AplicaInteresPorcentual(int tanto_porcentaje)

```
int main(){
    CuentaBancaria cuenta("20310381450100007510");
    CuentaBancaria otra_cuenta("20310381450100007511");

    // Las siguientes sentencias darían un error de compilación:

    // cuenta.SetIdentificador("20310381450100009876");
    // otra_cuenta.SetIdentificador("20310381450100003144");
```

En resumen:

Los constructores nos permiten ejecutar automáticamente un conjunto de instrucciones, cada vez que se crea un objeto.

Si el programador define cualquier constructor (con o sin parámetros) ya no está disponible el constructor de oficio.

Dentro de un constructor podemos ejecutar código y llamar a métodos de la clase.

IV.2.4.2. Constructores sin parámetros

Al igual que el programador puede definir un constructor con parámetros, también puede definir uno sin parámetros. Esto será útil cuando tengamos que hacer ciertas acciones iniciales al construir los objetos, pero no necesitamos pasar ninguna información como parámetro.

```
class MiClase{
    .....
public:
    MiClase(){          // Constructor añadido manualmente
        .....
    }
};

int main(){
    MiClase objeto;    // Se llama al constructor anterior
                      // Observe que aquí NO se ponen paréntesis:
                      // MiClase objeto(); <- INCORRECTO
```

Nota:

En C++, cualquier constructor sin parámetros (ya sea definido por el programador o por el propio lenguaje) se le denomina [constructor por defecto \(default constructor\)](#)

Veamos algunos ejemplos en los que es conveniente un constructor sin parámetros.

En un constructor sin parámetros podemos programar la inicialización de los datos miembro a un valor por defecto. Esto ya vimos cómo hacerlo en C++11 en la página 459. Ahora lo vemos de otra forma alternativa, con el constructor sin parámetros.

Para asignar a los datos miembro un valor por defecto lo podemos hacer de dos formas:

- ▷ ***O bien usamos la inicialización de C++ 11 en la declaración de los datos miembro.***
- ▷ ***O bien lo asignamos directamente en el constructor***

Ejemplo. Definimos un constructor en la clase `Circunferencia` para que, por defecto, cree la circunferencia goniométrica.

```
class Circunferencia{
private:
    double centro_x,  centro_y,  radio;
public:
    Circunferencia()          // Constructor sin parámetros
        :centro_x(0.0), centro_y(0.0), radio(1.0)
    {
    }
    .....
};

int main(){
    Circunferencia circ_goniometrica;  // 0.0, 0.0, 1
```

El anterior código es equivalente al siguiente, en el que se usa la inicialización de C++11 en la propia declaración del dato miembro (ver página 532):

```
class Circunferencia{
private:
    double centro_x = 0.0;
    double centro_y = 0.0;
    double radio = 1;
    .....
};

int main(){
    Circunferencia circ_goniometrica;  // 0.0, 0.0, 1
```

Si el valor por defecto a asignar a algún dato miembro no es un literal sino que se obtiene con algún procedimiento, lo programaremos dentro del constructor.

Ejemplo. Sobre la clase `Fecha` (ver página 515), definimos un constructor sin parámetros: su cometido será crear una fecha ligada al día actual.

Fecha	
- int	dia
- int	mes
- int	anio
- void	EsFechaCorrecta(int el_dia, int el_mes, int el_anio)
+	<code>Fecha()</code>
+ void	SetDiaMesAnio (int el_dia, int el_mes, int el_anio)
+ int	Dia()
+ int	Mes()
+ int	Anio()
+ 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;          // No hace falta entender este código
        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;
    }
    .....
};

int main(){
    Fecha fecha_hoy;                // Constructor sin parámetros

    cout << fecha_hoy.ToString();
```

Ejemplo. Supongamos que queremos generar números aleatorios entre 3 y 7. ¿Le parece que la siguiente secuencia es aleatoria?

5 5 5 6 6 6 7 7 7 3 3 3

Parece obvio que no. Existen algoritmos para generar secuencias de *números pseudo-aleatorios (pseudorandom numbers)*, es decir, secuencias de números con un comportamiento parecido al de una secuencia aleatoria.

Vamos a crear una clase para generar reales pseudo-aleatorios entre 0 y 1. En el constructor de la clase (que no tiene parámetros) programamos todo lo necesario para inicializar adecuadamente el generador de números pseudoaleatorios.

GeneradorAleatorioReales_0_1	
-
+	GeneradorAleatorioReales_0_1()
+	double Siguiente()

Cada vez que llamemos al método `Siguiente`, se generará el siguiente valor de la secuencia.

Del siguiente código, sólo tiene que entender las cabeceras del constructor y del método `Siguiente`:

```
#include <random> // para la generación de números pseudoaleatorios
#include <chrono> // para la semilla
using namespace std;

class GeneradorAleatorioReales_0_1{
private:
    mt19937 generador_mersenne; // Mersenne twister
    uniform_real_distribution<double> distribucion_uniforme;
```

```
long long Nanosec(){
    return
        chrono::high_resolution_clock::now().time_since_epoch().count();
}

public:
    GeneradorAleatorioReales_0_1(){
        distribucion_uniforme = uniform_real_distribution<double>(0.0, 1.0);
        const int A_DESCARTAR = 70000;
        // Panneton et al. ACM TOMS Volume 32 Issue 1, March 2006
        auto semilla = Nanosec();
        generador_mersenne.seed(semilla);
        generador_mersenne.discard(A_DESCARTAR);
    }
    double Siguiente(){
        return distribucion_uniforme(generador_mersenne);
    }
};

int main(){
    GeneradorAleatorioReales_0_1 aleatorio; // Constructor sin parámetros

    for (int i=0; i<100; i++)
        cout << aleatorio.Siguiente() << " ";

    cout << "\n\n";
}
```

http://decsai.ugr.es/jccubero/FP/IV_generador_aleatorio_0_1.cpp

IV.2.4.3. Sobrecarga de constructores

Todos los métodos de una clase se pueden sobrecargar, tal y como se vio con las funciones (ver página 446)

Lo mismo ocurre con los 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.

Podemos definir varios constructores. Esto nos permite crear objetos de maneras distintas, según nos convenga.

Si se desea un constructor sin parámetros, habrá que crearlo explícitamente, junto con los otros constructores.

Ejemplo. Retomamos el ejemplo de la clase `Fecha` (ver páginas 515 y 523). Incluimos los dos constructores dentro de la clase:

Fecha	
-	int dia
-	int mes
-	int anio
+	<code>Fecha()</code>
+	<code>Fecha(int el_dia, int el_mes, int el_anio)</code>


```
class Fecha{
private:
    int dia, mes, anio;
    .....
public:
    Fecha(){
        time_t hoy_time_t;
        .....
    }
    Fecha(int el_dia, int el_mes, int el_anio)
        :dia(el_dia),
        mes(el_mes),
        anio(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/jccubero/FP/IV_Fecha.cpp

Ejemplo. Sobre la cuenta bancaria, proporcionamos dos constructores:

- ▷ Un constructor que obligue a pasar el identificador pero no el saldo (en cuyo caso se quedará con cero)
- ▷ Otro constructor que obligue a pasar el identificador y el saldo

CuentaBancaria	
-	double saldo
-	double identificador
+	CuentaBancaria(string identificador_cuenta)
+	CuentaBancaria(string identificador_cuenta, double saldo_inicial)


```
class CuentaBancaria{
private:
    .....
public:
    CuentaBancaria(string identificador_cuenta)                // 1
        :identificador(identificador_cuenta)
    { }
    CuentaBancaria(string identificador_cuenta, double saldo_inicial) // 2
        :saldo(saldo_inicial),
        identificador(identificador_cuenta)
    { }
    .....
};

int main(){
    CuentaBancaria cuenta2("20310381450100007511");           // 1
        // cuenta2: 0, "20310381450100007511"
    CuentaBancaria cuenta1("20310381450100007510", 3000);     // 2
        // cuenta1: 3000, "20310381450100007510"
    // CuentaBancaria cuenta;      Error de compilación
```

IV.2.4.4. Llamadas entre constructores

Este apartado IV.2.4.4 es de ampliación. No entra en el examen.



Si observamos el ejemplo de la página 529, podemos apreciar que hay cierto código repetido en los constructores, ya que se realiza la misma asignación `identificador(identificador_cuenta)`. Lo resolvemos llamando a un constructor desde el otro constructor. Debe hacerse en la lista de inicialización del constructor:

Ejemplo. Llamamos a un constructor dentro del otro en el ejemplo de la cuenta bancaria.

```
class CuentaBancaria{
private:
    double saldo    = 0.0;
    string identificador;  ....
public:
    CuentaBancaria(string identificador_cuenta)                // 1
        :identificador(identificador_cuenta)
    { }
    CuentaBancaria(string identificador_cuenta, double saldo_inicial) // 2
        :CuentaBancaria(identificador_cuenta)
        // No puede haber nada más
    {
        saldo = saldo_inicial;
    }
    ....
};

int main(){
    CuentaBancaria cuenta2("20310381450100007511");            // 1
        // cuenta2: 0, "20310381450100007511"
    CuentaBancaria cuenta1("20310381450100007510", 3000);      // 2
        // cuenta1: 3000, "20310381450100007510"
```

IV.2.4.5. 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 euros;          // euros = ?  
cout << euros;     // Imprime un valor indeterminado
```

Lo mismo ocurre con un objeto que contenga datos miembro sin un valor determinado.

Ejemplo. Supongamos que no definimos constructores en la clase `Fecha`:

```
class Fecha {  
private:  
    int dia, mes, anio;  
    .....  
};  
int main(){  
    Fecha fecha_nacimiento_JC; // dia = mes = anio = ?  
                                // Estado inválido  
    cout << fecha_nacimiento_JC.Dia(); // Imprime basura
```



Diremos que un objeto se encuentra en un momento dado en un **estado inválido (invalid state)** si algunos de sus datos miembros **esenciales** no tienen un valor correcto. Cuando un objeto está en ese estado, se dice de forma informal que es un **zombi (zombie)**.

Consejo: *En la medida de lo posible, evite que durante la ejecución del programa, existan objetos en un estado inválido (zombies) en memoria*



Lo ideal sería conseguir que un objeto esté en un estado válido desde el mismo momento de su definición. Para ello, debemos asegurar que sus datos miembro tengan valores correctos. ¿Qué entendemos por correcto? Depende de cada clase. Desde luego, si es un valor indeterminado (no asignado previamente), será incorrecto.

En algunas clases, puede tener sentido asignar un valor por defecto (recuerde que los valores por defecto pueden especificarse o bien usando la inicialización de C++11 de los datos miembro -ver página 532-, o bien especificándolo en el constructor -ver página 521-)

- ▷ Una circunferencia por defecto podría ser la circunferencia goniométrica (centrada en el origen y de radio 1)

```
class Circunferencia{
private:
    double centro_x = 0.0;
    double centro_y = 0.0;
    double radio = 1;
    .....
};
int main(){
    Circunferencia circ_goniometrica; // 0.0, 0.0, 1
    .....
}
```



- ▷ Una cuenta bancaria se crea por defecto con saldo 0.

```
class CuentaBancaria{
private:
    double saldo = 0.0;
    .....
}
```



Sin embargo, los valores por defecto no resuelven el problema en ejemplos como los siguientes:

- ▷ **¿Qué identificador por defecto asociamos a una nueva cuenta bancaria? No vale "". Debe ser un identificador válido.**

```
class CuentaBancaria{
private:
    double saldo = 0.0;
    string identificador = "";
    .....
};
int main(){
    CuentaBancaria cuenta; // 0.0, ""
```



- ▷ **Si tenemos la clase Persona, ¿qué nombre o DNI se le asignan por defecto? No tiene sentido.**

```
class Persona{
private:
    string nombre = "Juan Carlos"; //
    .....
};
int main(){
    Persona una_persona; // "Juan Carlos"
```



Para resolverlo, podemos definir un constructor y obligar a que se pasen como parámetros actuales los valores iniciales de los datos miembro.

Ejemplo. Retomamos la clase CuentaBancaria (página 529)

```
class CuentaBancaria{
private:
    double saldo = 0.0;    // Tiene sentido un valor por defecto
    string identificador;  // No tiene sentido un valor por defecto
public:
    CuentaBancaria(string identificador_cuenta)
        :identificador(identificador_cuenta)
    { }
    .....
};

int main(){

    // CuentaBancaria cuenta;  <-- Error de compilación
    CuentaBancaria cuenta("20310381450100007511");
        // cuenta: 0, "20310381450100007511"
        // El objeto cuenta está en un estado válido
```



Ejemplo. Clase Persona:

```
class Persona{
private:
    string nombre; // No tiene sentido un valor por defecto
    .....
public:
    Persona(string nombre_persona, string DNI_persona)
        :nombre(nombre_persona), DNI(DNI_persona)
    { }
    .....
};

int main(){

    // Persona una_persona; <-- Error de compilación
    Persona una_persona("Juan Carlos", "29031456Y");
        // una_persona: "Juan Carlos", "29031456Y"
        // El objeto una_persona está en un estado válido
```



Así pues, los constructores nos ayudan a que el objeto esté en un estado válido en el mismo momento de su definición. Ahora bien, se nos pueden plantear situaciones problemáticas como la siguiente:

Ejemplo. Retomemos el ejemplo de la página 510. Si no queremos que un objeto de la clase `Fecha` esté en un estado inválido en ningún momento, deberíamos escribir el siguiente código:

```
while (hay_datos){
    cin >> dia >> mes >> anio;
    hay_datos = (anio != -1);

    if (hay_datos){
        Fecha fecha_nacimiento(dia, mes, anio);
        cout << fecha_nacimiento.ToString();
    }
}
```

Pero esto nos obliga a crear un objeto nuevo en cada iteración. Tal vez estemos interesados en guardar todos esos objetos, pero si no es así, entonces podemos aceptar en aras de la eficiencia que sería mejor crear un único objeto *vacío* fuera del bucle. Estaría en un estado inválido hasta que no se ejecutase el método `SetDiaMesAnio` dentro del bucle.

```
Fecha fecha_nacimiento; // Se necesita un constructor sin parámetros

while (hay_datos){
    cin >> dia >> mes >> anio;
    hay_datos = (anio != -1);

    if (hay_datos){
        fecha_nacimiento.SetDiaMesAnio(dia, mes, anio);
        cout << fecha_nacimiento.ToString();
    }
}
```


En resumen:

Debemos evitar, en la medida de lo posible, que un objeto esté en un estado inválido (zombie) en algún momento de la ejecución del programa. Para ello:

- ▷ ***Cuando el objeto se construya por primera vez, podemos asignar valores por defecto específicos a los datos miembro. Esto lo haremos sólo cuando tenga sentido: saldo a 0 por defecto en una cuenta bancaria, circunferencia goniométrica por defecto, etc.***
- ▷ ***Si no tiene sentido usar valores por defecto, podemos definir un constructor y pasarle como parámetros actuales los valores de los datos miembro.***
Para simplificar el código de las clases, a lo largo de la asignatura asumiremos que dichos parámetros actuales serán correctos
- ▷ ***En cualquier caso, con las herramientas vistas hasta ahora, no siempre será posible resolver este problema y a veces tendremos que aceptar que un objeto pueda estar temporalmente en un estado inválido.***

IV.2.5. Comprobación y tratamiento de las precondiciones

¿Qué hacemos si los parámetros actuales a un método/función/constructor no son correctos?

En la página 404 se vieron las precondiciones de una función y en la página 499 vimos el tratamiento de las precondiciones en los métodos de una clase. Ahora vamos a ver con más detalle el tratamiento de precondiciones para cualquier tipo de método o función. Nos planteamos 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?

Empezaremos viendo soluciones para cualquier tipo de método y luego trataremos el caso de los constructores.

IV.2.5.1. Comprobar o no comprobar

¿Cuándo debemos comprobar las precondiciones? Vamos a intentar dar respuesta a esta pregunta viendo algunos ejemplos.

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 Circunferencia.

```
const double PI = 6 * asin(0.5);

class Circunferencia{
private:
    double centro_x = 0.0;
    double centro_y = 0.0;
    double radio      = 1.0;    // Circunferencia goniométrica
public:
    void SetRadio(double el_radio){
        if (radio >= 0.0)
            radio = el_radio;
    }
    .....
};
```

El método SetRadio espera recibir un valor positivo. Si no es así, no deberíamos asignarlo al radio. Por tanto, conviene comprobar la precondición con el correspondiente if.

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;
    }
};
```

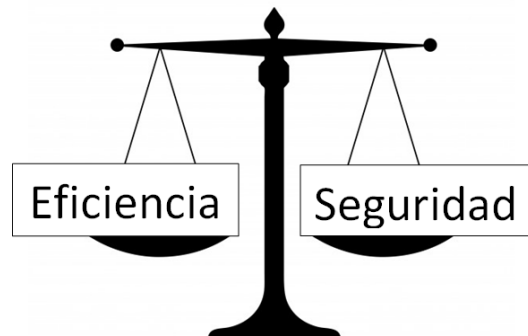
En este caso, es mejor comprobar la precondición ya que no podemos permitir un saldo negativo.

En resumen:

¿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?

IV.2.5.2. Tratamiento de la violación de la precondición

Supongamos que decidimos comprobar las precondiciones de una función/método y resulta que no se satisfacen ¿Qué hacemos? Posibilidades:

▷ **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; // poco recomendable
    }
```



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.

¿Y cómo sabemos que se ha violado la precondición desde el sitio en el que se invoca al método/función? ¿Lo siguiente sería correcto?

```
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 (o pasando un parámetro por referencia -se verá en el segundo cuatrimestre-)
- ▷ Mucho mejor: lanzar una **excepción (exception)** (ver apartado de Ampliación en el tema V)

Veamos cómo realizar la notificación de un error a la *antigua usanza*. Será 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. Si no puede elegirse tal valor, se puede pasar un parámetro por referencia (se verá en el segundo cuatrimestre)
- ▷ 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 `Ingresa` para que devuelva el código de error:

```
void Ingresa(double cantidad) --> bool Ingresa(double cantidad)

class CuentaBancaria{
.....
    bool Ingresa(double cantidad){
        bool error;

        error = (cantidad < 0);

        if (!error)
            saldo = saldo + cantidad;

        return error;
    }
};

int main(){
    .....
    bool error_cuenta;
    error_cuenta = cuenta.Ingresa(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 precondición. Esto también es válido para el constructor.

En resumen:

*Si decidimos que es necesario **comprobar** alguna precondición dentro de un método/función y ésta se viola, ¿qué debemos hacer como respuesta?*

- ▷ *O bien no hacemos nada 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. La notificación jamás se hará imprimiendo un mensaje en pantalla desde dentro del método/función.*
- ▷ *En el tema V se verá un apartado de Ampliación en el que se muestra cómo resolver apropiadamente los dos problemas anteriores usando el mecanismo de las excepciones.*

Nota:

En la página 555 se ve una extensión del resumen anterior incluyendo los constructores (en un apartado de Ampliación)

IV.2.5.3. Comprobación de las precondiciones en el constructor

Este apartado IV.2.5.3 es de ampliación. No entra en el examen



En la página 535 nos habíamos planteado la siguiente pregunta: ¿Qué hacemos si los valores pasados como parámetros actuales al constructor no son correctos?

Ejemplo. Retomamos el ejemplo de la clase `Fecha`

El constructor asignaba directamente los parámetros actuales a los datos miembro. ¿Qué ocurre si los datos son incorrectos? El objeto se queda en un estado inválido:

```
public:
class Fecha{
    .....
    Fecha(int el_dia, int el_mes, int el_anio)
        :dia(el_dia), mes(el_mes), anio(el_anio)
    .....
};

int main(){
    Fecha una_fecha(-1, 2, -9); // ?, ? ,?

    // una_fecha está en un estado inválido -zombie-
```



La forma correcta de resolver este problema es usar excepciones (se ve como ampliación en el tema V). Así pues, si los parámetros actuales pasados al constructor no son correctos, se lanza una excepción y no se permite la creación del objeto. Como no sabemos por ahora cómo hacerlo, al menos podemos paliar algo el problema planteado asignándole un valor concreto a los datos miembro que nos indique que ha habido un problema. El objeto seguirá en un estado inválido pero **reconocible**

- ▷ A un objeto de la clase `Fecha`, le asignaríamos `-1, -1, -1`.
- ▷ A un objeto de la clase `Circunferencia`, le asignaríamos al radio `-1`. Al centro le podríamos asignar `(NAN,NAN)`.
- ▷ A un objeto de la clase `CuentaBancaria`, le asignaríamos `""` al identificador.
- ▷ A un objeto de la clase `SegmentoDirigido`, le asignaríamos `NAN` a las 4 coordenadas.

La asignación de dicho valor la podemos hacer como cualquier valor por defecto (ver página 521):

- ▷ Usando la inicialización de los datos miembro de C++11.
- ▷ Haciéndolo dentro del constructor.

Si decidimos hacerlo de la primera forma, recuerde lo visto en la página 512: si la clase ya dispone de un constructor con parámetros, entonces ya no está disponible el constructor de oficio. Por lo tanto, tendremos que proporcionar un constructor adicional sin parámetros que no haga nada. Veámoslo:

Ejemplo. Clase Circunferencia. Si no pasamos parámetros al constructor queremos que se cree, por defecto, la circunferencia goniométrica.

Opciones:

- ▷ **O bien usamos la inicialización de C++ 11 en la declaración de los datos miembro:**

```
class Circunferencia{
private:
    double centro_x = 0.0;
    double centro_y = 0.0;
    double radio     = 1.0;

    EsRadioCorrecto(double r){
        return r > 0;
    }
public:
    Circunferencia(){
        // No hacemos nada => Se quedan los valores por defecto
        // (Circunferencia goniométrica)
    }
    Circunferencia(double abscisa, double ordenada, double el_radio){
        if (EsRadioCorrecto(el_radio)){
            SetRadio(el_radio);
            SetCentro(abscisa, ordenada);
        }
        else{
            centro_x = NAN;
            centro_y = NAN;
            radio = -1;
        }
    }
    void SetCentro(double abscisa, double ordenada){
```

```
        centro_x = abscisa;
        centro_y = ordenada;
    }
    void SetRadio(double el_radio){
        if (EsRadioCorrecto(el_radio))
            radio = el_radio;
    }
    .....
};
int main(){
    Circunferencia circ_goniometrica; // 0.0, 0.0, 1.0
    Circunferencia circ_estado_invalido(1.0, 2.0, -1.0); // NAN, NAN, -1
    Circunferencia circ_estado_valido(1.0, 2.0, 1.0); // 1.0, 2.0, 1.0
    .....
}
```

Observe que hemos definido un constructor sin parámetros que no hace nada. Esto es necesario ya que si sólo dejamos el constructor con parámetros, ya no está disponible el constructor de oficio y no podríamos crear una circunferencia sin pasar parámetros.

▷ **O bien lo asignamos en el constructor sin parámetros:**

```
class Circunferencia{
private:
    double centro_x;
    double centro_y;
    double radio;

    EsRadioCorrecto(double r){
        return r > 0;
    }
public:
    Circunferencia()
```

```
        :centro_x(0.0), centro_y(0.0), radio (1.0)
    {
    }
    Circunferencia(double abscisa, double ordenada, double el_radio){
        // No cambia
    }
    // El resto de métodos no cambia
    .....
};
int main(){
    Circunferencia circ_goniometrica; // 0.0, 0.0, 1.0
    Circunferencia circ_estado_invalido(1.0, 2.0, -1.0); // NAN, NAN, -1
    Circunferencia circ_estado_valido(1.0, 2.0, 1.0); // 1.0, 2.0, 1.0
    .....
}
```


Ejemplo. Retomamos el ejemplo de la clase `Fecha` (página 528).

Fecha	
- int	dia
- int	mes
- int	anio
- bool	EsFechaCorrecta(int el_dia, int el_mes, int el_anio)
+	Fecha(int el_dia, int el_mes, int el_anio)
+	Fecha()
+ void	SetDiaMesAnio(int el_dia, int el_mes, int el_anio)
+ int	Dia()
+ int	Mes()
+ int	Anio()
+ string	ToString()



Opciones:

- ▷ **O bien usamos la inicialización de C++ 11 en la declaración de los datos miembro:**

```
class Fecha{
    int dia  = -1;
    int mes  = -1;
    int anio = -1;

    Fecha(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;
        }
    }
    .....
}
```

▷ O bien lo asignamos en el constructor:

```
class Fecha {
    int dia;
    int mes;
    int anio;

    Fecha(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;
        }
        else
            // lo mejor sería lanzar una excepción
            dia = mes = anio = -1;
    }
    .....
}
```

En cualquiera de los dos casos, el programa principal quedaría así:

```
int main(){
    Fecha una_fecha(-1, 2, -9);                // -1, -1 , -1
    // una_fecha está en un estado inválido, pero "reconocible"

    Fecha nacimiento_JC (27, 2, 1967);        // 27, 2, 1967
    // estado válido

    nacimiento_JC.SetDiaMesAnio(-1, 2, 100);  // 27, 2, 1967
    // se queda como estaba. Estado válido.
```



http://decsai.ugr.es/jccubero/FP/IV_Fecha_Ampliacion.cpp

En resumen, ampliando lo visto en la página 547, tendríamos:

*Si decidimos que es necesario **comprobar** alguna precondición dentro de un método/función/constructor y ésta se viola, ¿qué debemos hacer como respuesta?*

- ▷ *O bien no hacemos nada 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. Esta solución no es aplicable a los constructores ya que éstos no pueden devolver ningún valor.*
La notificación jamás se hará imprimiendo un mensaje en pantalla desde dentro del método/función/constructor.
- ▷ *En el tema V se verá un apartado de Ampliación en el que se muestra cómo resolver apropiadamente los dos problemas anteriores usando el mecanismo de las excepciones.*

IV.2.6. Registros (structs)

IV.2.6.1. El tipo de dato struct

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 dato tipo *registro (record)* .

Un registro permite almacenar varios elementos de (posiblemente) distintos tipos y gestionarlos como uno sólo. Cada uno de los datos agrupados en un registro es un *campo* (son como los datos miembro públicos de las clases)

Los campos de un registro se suponen relacionados, que hacen referencia a una misma entidad. Cada campo tiene un *nombre*.

En C++, los registros se denominan *struct* .

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 {  
    double abscisa;  
    double 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 `double`.

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 = 4.0;
un_punto.ordenada = 6.0;
```

Es posible declarar variables `struct` junto a la definición del tipo (aunque normalmente lo haremos de forma separada):

```
struct Punto2D {
    double abscisa;
    double 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 `double`.

IV.2.6.2. Funciones/métodos y el tipo `struct`

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 PuntoMedio (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);  
}
```

IV.2.6.3. 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 `PuntoMedio` 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 `PuntoMedio` 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 (hay_datos_por_procesar) {
    Punto2D punto;

    // punto sólo es accesible dentro del ciclo while.
    // Se crea un struct nuevo en cada iteración y "desaparece"
    // el de la anterior.
    // Evítelo por la recarga computacional ocasionada
}
```

IV.2.6.4. 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`.

IV.2.7. Datos miembro constantes

Recuperamos el ejemplo de la cuenta bancaria. Recuerde 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.

Las constantes a nivel de objeto presentan algunas dificultades (ver Tema V) Por lo tanto, durante la asignatura, fomentaremos el uso de constantes estáticas en detrimento de las constantes a nivel de objeto.

IV.2.7.1. 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 la lista de inicialización del constructor. Son construcciones del tipo `<dato_miembro> (<valor inicial>)` separadas por coma. 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 string CTE_STRING;
public:
    MiClase(double un_real, int un_string)
        :CTE_REAL(un_real) ,    // Correcto. Aquí se le asigna el valor
        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. Recupere la clase `CuentaBancaria` y defina 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);
    .....
```

IV.2.7.2. Constantes a nivel de clase (estáticas)

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.

*El valor de las constantes estáticas se asignan durante la compilación. Por tanto, está permitido usar datos miembro constantes **públicos***

Constantes estáticas NO enteras

La forma de definir las constantes estáticas no enteras es algo farragoso. Lo explicamos a continuación, pero no hay que aprendérselo de memoria para el examen.

- ▷ 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 separada (separate compilation)* . Se verá con más detalle en el segundo cuatrimestre.

Ampliación:

En c++11 también se permite declarar una constante estática de la siguiente forma:

```
static constexpr double CTE_REAL = 4.0;
```



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;  
    .....  
};
```

IV.2.8. Vectores y objetos

A lo largo de la asignatura seremos capaces de:

- ▷ Definir un vector dentro de un método de un objeto.
- ▷ Incluir un vector como un dato miembro de un objeto.
- ▷ Definir un vector en el que cada componente sea un objeto.

Empezamos viendo los dos primeros puntos y posteriormente trataremos el último.

IV.2.8.1. Los vectores como datos locales de un método

- ▷ Un vector puede ser un dato local de un método.
- ▷ Su tamaño se debe especificar con una constante definida dentro del método o con una constante definida en un ámbito superior.
- ▷ El vector se creará en la pila, en el momento de ejecutarse el método. Cuando éste termine, se libera la memoria asociada al vector. Lo mismo ocurre si fuese local a una función.
- ▷ Si se necesita inicializar el vector, se hará tal y como se indicó en la página 315

```
class MiClase{
    .....
    void Metodo(){
        const int TAMANIO = 30; // static cuando es dato miembro
        char vector[TAMANIO];
        .....
    }
};

int main(){
    MiClase objeto;
    objeto.Metodo() // <- Se crea el vector local con 30 componentes
    .....        // Terminado el método, se destruye el vector local
```

Las funciones o métodos pueden definir vectores como datos locales para hacer sus cálculos. Pero no pueden devolverlos.

Lo que sí podremos hacer es devolver objetos que contengan vectores.

Ejemplo. Clase Fecha. Este ejemplo ya se vio en la página 528

```
class Fecha{
private:
    int dia, mes, anio;

    bool EsCorrecta(int el_dia, int el_mes, int el_anio){
        .....
        const int dias_por_mes[12] =
            {31,28,31,30,31,30,31,31,30,31,30,31};

        es_fecha_correcta = ..... &&
                           el_dia <= dias_por_mes[el_mes - 1] &&
                           .....

        .....
        return es_fecha_correcta;
    }
    .....
}
```

IV.2.8.2. Los vectores como datos miembro

Los vectores de C++ son potentes pero pueden provocar errores de ejecución si accedemos a componentes no reservadas. Por ello, definiremos nuestras propias clases para trabajar con vectores de forma segura.

- ▷ Para usar un vector clásico como dato miembro de una clase, debemos dimensionarlo o bien con un literal, o bien con una constante global definida fuera de la clase, o bien con una constante definida dentro de la clase (en este caso, debe ser estática)
- ▷ Cuando se crea un objeto, se creará automáticamente el vector. El vector existirá mientras exista el objeto que lo contiene.
- ▷ Todo lo anterior también se aplica si en vez de tener un vector tenemos una matriz como dato miembro.
- ▷ Por ahora trabajamos con vectores de tipos simples. En el próximo tema veremos los vectores (o matrices) de objetos.

```
class MiClase{
private:
    static const int TAMANIO = 30;
    char vector[TAMANIO];
    .....
};

int main(){
    MiClase objeto; // Se crea una zona de memoria reservada
                   // para el objeto. Éste contiene un
                   // vector privado con 30 componentes
```

Dentro de la clase definiremos los métodos que acceden a las componentes del vector. Éstos establecerán la política de acceso a dichas componentes, como por ejemplo, si permitimos modificar cualquier componente en cualquier momento o si sólo permitimos añadir.

Ejemplo. Retomamos el ejemplo del autobús de la página 327 y construimos la clase `Autobus`

- ▷ El nombre del conductor será un parámetro a pasar en el constructor.
- ▷ Definimos un método `Situa` indicando el nombre del pasajero y el del asiento que se le va a asignar:

```
void Situa(int asiento, string nombre_pasajero)
```

Autobus	
- const int	<code>MAX_PLAZAS</code>
- string	<code>pasajeros[MAX_PLAZAS]</code>
- int	<code>numero_actual_pasajeros</code>
+	<code>Autobus(string nombre_conductor)</code>
+ void	<code>Situa(int asiento, string nombre_pasajero)</code>
+ string	<code>Pasajero(int asiento)</code>
+ string	<code>Conductor()</code>
+ bool	<code>Ocupado(int asiento)</code>
+ int	<code>Capacidad()</code>
+ int	<code>NumeroActualPasajeros()</code>

Esta clase se usará en la siguiente forma:

```
int main(){
    .....
    Autobus alsa(conductor);
    .....
    while (hay_datos_por_leer){
        alsa.Situa(asiento, nombre_pasajero);
        .....
    }
}
```

Veamos el programa completo:

```
int main(){
    // En la siguiente versión las cadenas introducidas
    // no deben tener espacios en blanco

    const string TERMINADOR = "-";
    string conductor,
           nombre_pasajero;
    int    asiento,
           capacidad_autobus;

    cout << "Autobús.\n";
    cout << "\nIntroduzca nombre del conductor: ";
    cin >> conductor;

    Autobus alsa(conductor);

    cout << "\nIntroduzca los nombres de los pasajeros y su asiento."
          << "Termine con " << TERMINADOR << "\n";
    cout << "\nNombre: ";
    cin >> nombre_pasajero;

    while (nombre_pasajero != TERMINADOR){
        cout << "Asiento: ";
        cin >> asiento;

        alsa.Situa(asiento, nombre_pasajero);

        cout << "\nNombre: ";
        cin >> nombre_pasajero;
    }

    cout << "\nTotal de pasajeros: " << alsa.NumeroActualPasajeros();
    cout << "\nConductor: " << alsa.Conductor() << "\n";
}
```

```
capacidad_autobus = alsa.Capacidad();

for (int i=1; i < capacidad_autobus; i++){
    cout << "\nAsiento número: " << i;

    if (alsa.Ocupado(i))
        cout << " Pasajero: " << alsa.Pasajero(i);
    else
        cout << " Vacío";
}
}
```

Implementamos ahora la clase. Observe cómo ocultamos información al situar dentro de la clase las constantes MAX_PLAZAS y VACIO.

```
class Autobus{
private:
    static const string VACIO;
    static const int MAX_PLAZAS = 50;
    string pasajeros[MAX_PLAZAS];
    int numero_actual_pasajeros;
public:
    Autobus(string nombre_conductor){
        pasajeros[0] = nombre_conductor;
        numero_actual_pasajeros = 0; // El conductor no cuenta como pasajero

        for (int i=1; i < MAX_PLAZAS; i++)
            pasajeros[i] = VACIO;
    }
    void Situa(int asiento, string nombre_pasajero){
        if (asiento >= 1 && asiento < MAX_PLAZAS){
            if (! Ocupado(asiento)){
                pasajeros[asiento] = nombre_pasajero;
            }
        }
    }
};
```

```
        numero_actual_pasajeros++;
    }
}

// Prec: 0 < asiento < MAX_PLAZAS
string Pasajero(int asiento){
    return pasajeros[asiento];
}

string Conductor(){
    return pasajeros[0];
}

// Prec: 0 < asiento < MAX_PLAZAS
bool Ocupado(int asiento){
    return pasajeros[asiento] != VACIO;
}

int Capacidad(){
    return MAX_PLAZAS;
}

int NumeroActualPasajeros(){
    return numero_actual_pasajeros;
}

};

const string Autobus::VACIO = "";
```

http://decsai.ugr.es/jccubero/FP/IV_autobus_clases.cpp

Ampliación:

Si se desea, el método **Situa** podría devolver un **bool** indicando si se ha podido situar correctamente al pasajero. En cualquier caso, si queremos notificar problemas durante la ejecución de un método es mejor hacerlo con el mecanismo de las excepciones tal y como se indica en la página 544



Ejemplo. Retomamos el ejemplo de la página 325 creando la clase `CalificacionesFinales`.

- ▷ Añadiremos el nombre del alumno junto con su nota.
- ▷ Internamente usaremos un vector para almacenar los nombres y otro vector para las notas.
- ▷ Al contrario del ejemplo del autobús, no dejaremos huecos, sólo permitimos añadir datos y se irán llenando las componentes desde el inicio.

	utilizados = 4							
notas	X	X	X	X	?	?	...	?
nombres	X	X	X	X	?	?	...	?

Para forzar esta situación, sólo vamos a permitir insertar nuevos datos a través de un único método, a saber:

```
void Aniade(string nombre_alumno, double nota_alumno)
```

Observe que no pasamos como parámetro el índice de la componente a modificar. El método `Aniade` siempre añade al final.

- ▷ Una vez añadido un alumno con su nota, no se puede borrar posteriormente.

CalificacionesFinales	
- const int	MAX_ALUMNOS
- double	notas[MAX_ALUMNOS]
- string	nombres[MAX_ALUMNOS]
- int	utilizados
+	Notas()
+ int	Capacidad()
+ int	TotalAlumnos()
+ double	CalificacionAlumno(int posicion)
+ string	NombreAlumno(int posicion)
+ void	Aniade(string nombre_alumno, double nota_alumno)
+ double	MediaGlobal()
+ int	SuperanMedia()

Esta clase se usará en la siguiente forma:

```
int main(){
    Notas notas_FP;
    .....

    while (hay_datos_por_leer){
        .....
        notas_FP.Aniade(nombre, nota);
    }
}
```

Veamos el programa completo:

```
#include <iostream>
#include <string>

using namespace std;

class CalificacionesFinales{
private:
    static const int MAX_ALUMNOS = 100;
    double notas[MAX_ALUMNOS];
```



```
    string nombres[MAX_ALUMNOS];
    int utilizados;
public:
    CalificacionesFinales()
        :utilizados(0){
    }
    int Capacidad(){
        return MAX_ALUMNOS;
    }
    int TotalAlumnos(){
        return utilizados;
    }
    double CalificacionAlumno(int posicion){
        return notas[posicion];
    }
    string NombreAlumno(int posicion){
        return nombres[posicion];
    }
    void Aniade(string nombre_alumno, double nota_alumno){
        if (utilizados < MAX_ALUMNOS){
            notas[utilizados] = nota_alumno;
            nombres[utilizados] = nombre_alumno;
            utilizados++;
        }
    }
    double MediaGlobal(){
        double media = 0;

        for (int i = 0; i < utilizados; i++)
            media = media + notas[i];

        media = media / utilizados;
```

```
        return media;
    }
    int SuperanMedia(){
        double media = MediaGlobal();
        int superan_media = 0;

        for (int i = 0; i < utilizados; i++){
            if (notas[i] > media)
                superan_media++;
        }

        return superan_media;
    }
};

int main(){
    const string TERMINADOR = "-";
    int capacidad;
    string nombre;
    double nota, media, superan_media;
    CalificacionesFinales notas_FP;

    cout << "Introduzca nombre de alumno y su nota. "
         << TERMINADOR << " para terminar\n";

    capacidad = notas_FP.Capacidad();
    cin >> nombre;

    while (nombre != TERMINADOR && notas_FP.TotalAlumnos() < capacidad){
        cin >> nota;
        notas_FP.Aniade(nombre, nota);
        cin >> nombre;
    }
}
```

```
media = notas_FP.MediaGlobal();  
superan_media = notas_FP.SuperanMedia();  
  
cout << "Media Aritmetica = " << media << "\n";  
cout << superan_media << " alumnos han superado la media\n";  
}
```

http://decsai.ugr.es/jccubero/FP/IV_notas_clases.cpp

IV.2.8.3. Los vectores como parámetros a un método

Los vectores clásicos que estamos viendo pueden pasarse como parámetros a una función o método. Sin embargo, no lo veremos en esta asignatura porque:

- ▷ En C++, al pasar un vector como parámetro, únicamente se pasa una copia de la dirección de memoria inicial de las componentes, por lo que desde dentro de la función o método podemos modificarlas.

Esto rompe el concepto de *paso de parámetro por valor*.

- ▷ Los vectores son una herramienta potente pero peligrosa si no se usan con cuidado. Por ello, se pretende acostumbrar al alumno a que construya sus propias clases si necesita almacenar varios valores. Serán los objetos de estas clases los que pasaremos como parámetro a otros métodos.

En esta asignatura no pasaremos los vectores como parámetros de las funciones o métodos.

IV.2.8.4. Comprobación de las precondiciones trabajando con vectores

Recuerde lo indicado en la página 542:

Si la violación de una precondición de un método *público* puede provocar errores de ejecución graves, dicha precondición debe comprobarse dentro del método. En otro caso, puede omitirse.

Al trabajar con un vector dato miembro de una clase, si un método modifica componentes de dicho vector, entonces comprobaremos las precondiciones dentro del método.

Ejemplo. En el ejemplo del autobús, hemos comprobado la precondition en el método `Situa` ya que si se viola y pasamos como parámetro un índice de asiento fuera del rango admitido, se podría modificar una zona de memoria no reservada, con consecuencias imprevisibles.

▷ **Versión sin comprobar la precondition:**

```
// Prec: 0 < asiento < MAX_PLAZAS

void Situa(int asiento, string nombre_pasajero){
    if (! Ocupado(asiento)){
        pasajeros[asiento] = nombre_pasajero;
        numero_actual_pasajeros++;
    }
}
```



▷ **Versión comprobando la precondition:**

```
void Situa(int asiento, string nombre_pasajero){

    if (asiento >= 1 && asiento < MAX_PLAZAS){
        if (! Ocupado(asiento)){
            pasajeros[asiento] = nombre_pasajero;
            numero_actual_pasajeros++;
        }
    }
}
```



Por contra, si el índice pasado a `Pasajero` no es correcto, simplemente devolverá un valor basura, pero no corremos el peligro de modificar una componente no reservada. Por lo tanto, podemos evitar la comprobación de la precondition dentro del método `Pasajero`. Lo mismo ocurre con el método `Ocupado`.

En cualquier caso, recordemos que si accedemos a una componente no reservada, aunque no se modifique, el programa puede terminar abruptamente ya que C++ indica que el comportamiento es indeterminado.

Ejemplo. En el ejemplo de las notas, hemos comprobado la precondition en el método `Aniade` ya que, si se viola e intentamos añadir por encima de la zona reservada, las consecuencias son imprevisibles.

▷ **Versión sin comprobar la precondition:**

```
// Prec: 0 <= utilizados < MAX_ALUMNOS

void Aniade(string nombre_alumno, double calificacion_alumno){
    nota[utilizados] = calificacion_alumno;
    nombre[utilizados] = nombre_alumno;
    utilizados++;
}
```



▷ **Versión comprobando la precondition:**

```
void Aniade(string nombre_alumno, double calificacion_alumno){
    if (utilizados < MAX_ALUMNOS){
        nota[utilizados] = calificacion_alumno;
        nombre[utilizados] = nombre_alumno;
        utilizados++;
    }
}
```



Al no modificar componentes, hemos decidido no comprobar las condiciones en los métodos `CalificacionAlumno` y `NombreAlumno`

IV.2.9. La clase Secuencia de caracteres

IV.2.9.1. Métodos básicos

Ya sabemos que hay que trabajar con cuidado con los vectores con corchetes. Nos gustaría crear una clase genérica `MiVector` que protegiese adecuadamente las componentes de una incorrecta manipulación.

C++ proporciona clases específicas para trabajar de forma segura con vectores genéricos de datos como por ejemplo la plantilla `vector` de la STL. Se verán en el segundo cuatrimestre.

Con las herramientas que conocemos hasta ahora, vamos a tener que crear una clase para cada tipo de dato que queramos manejar: `MiVectorEnteros`, `MiVectorDoubles`, etc.

En lo que sigue nos ceñiremos al tipo de dato `char`. Vamos a construir una clase a la que denominaremos `SecuenciaCaracteres` de forma que:

- ▷ Represente una secuencia consecutiva de caracteres sin huecos.
- ▷ Internamente, usaremos un vector de tipo `char`:
- ▷ Los métodos de la interfaz pública de la clase protegerán convenientemente el acceso a las componentes del vector.

Por lo tanto, no permitiremos operaciones (como añadir o modificar componentes) que impliquen dejar huecos.

`total_utilizados = 4`

X	X	X	X	?	?	...	?
---	---	---	---	---	---	-----	---

Una primera versión:

SecuenciaCaracteres	
- const int	<u>TAMANIO</u>
- char	vector_privado[TAMANIO]
- int	total_utilizados
+	SecuenciaCaracteres()
+ int	TotalUtilizados()
+ int	Capacidad()
+ void	Aniade(char nuevo)
+ void	Modifica(int indice, char nuevo)
+ char	Elemento(int indice)

- ▷ TotalUtilizados **devuelve el número de componentes ocupadas.**
- ▷ Capacidad **devuelve el tamaño del vector (número de componentes reservadas en memoria)**
- ▷ Aniade **añade al final un valor más.**
- ▷ Modifica **modifica el valor de una componente ya existente (previamente añadida a través del método Aniade).**

Los métodos Aniade y Modifica pueden modificar componentes del vector. Para impedir que se acceda a componentes no reservadas, comprobaremos la precondition de que la posición esté en el rango correcto.

- ▷ Elemento **devuelve el valor de la componente en el índice señalado.**

Un ejemplo de uso de esta clase:

```
int main(){
    SecuenciaCaracteres secuencia;
    int num_elementos;

    secuencia.Aniade('h');    // -> h
    secuencia.Aniade('o');    // -> ho
    secuencia.Aniade('l');    // -> hol
    secuencia.Aniade('a');    // -> hola

    num_elementos = secuencia.TotalUtilizados()

    for (int i = 0; i < num_elementos; i++)
        cout << secuencia.Elemento(i) << " ";

    secuencia.vector_privado[1] = 'g'; // Error compilac. Dato privado. 😊
    secuencia.vector_privado[70] = 'g'; // Error compilac. Dato privado.

    secuencia.Modifica(9,'l'); // -> hola    Acceso incorrecto
    secuencia.Modifica(0,'l'); // -> lola
```

Definamos los métodos:

```
class SecuenciaCaracteres{
private:
    static const int TAMANIO = 50;
    char vector_privado[TAMANIO];
    int total_utilizados;
public:
    SecuenciaCaracteres()
        :total_utilizados(0) {
    }
    int TotalUtilizados(){
        return total_utilizados;
    }
    int Capacidad(){
        return TAMANIO;
    }
    void Aniade(char nuevo){
        if (total_utilizados < TAMANIO){
            vector_privado[total_utilizados] = nuevo;
            total_utilizados++;
        }
    }
    void Modifica(int posicion, char nuevo){
        if (posicion >= 0 && posicion < total_utilizados)
            vector_privado[posicion] = nuevo;
    }
    char Elemento(int indice){
        return vector_privado[indice];
    }
};
```

http://decsai.ugr.es/jccubero/FP/IV_secuencia_base.cpp

¿Debería añadirse el siguiente método?

```
class SecuenciaCaracteres{  
    .....  
    void SetTotalUtilizados(int nuevo_total){  
        total_utilizados = nuevo_total;  
    }  
};
```



Obviamente no. La gestión de `total_utilizados` es responsabilidad de la clase, y ha de actualizarse automáticamente dentro de la clase y no de forma manual. Un par de excepciones se ven en los siguientes ejemplos.

***Ejercicio.* Definir un método para *borrar* todos los caracteres.**

```
class SecuenciaCaracteres{  
    .....  
    void EliminaTodos(){  
        total_utilizados = 0;  
    }  
};
```



***Ejercicio.* Definir un método para *borrar* el último carácter.**

```
class SecuenciaCaracteres{  
    .....  
    void EliminaUltimo(){  
        total_utilizados--;  
    }  
};
```



Ejercicio. Incluso sería aceptable definir un método para *truncar* a partir de una posición. Pero siempre sin dejar huecos.

```
class SecuenciaCaracteres{  
    .....  
    void Trunca(int nuevo_total){  
        if (0 <= nuevo_total && nuevo_total < total_utilizados)  
            total_utilizados = nuevo_total;  
    }  
};
```



Si queremos una secuencia de reales:

```
class SecuenciaDoubles{
private:
    static const int TAMANIO = 50;
    double vector_privado[TAMANIO];
    int total_utilizados;
public:
    SecuenciaDoubles()
        :total_utilizados(0)    {   }

    < Los métodos TotalUtilizados() y Capacidad() no varían >

    void Aniade(double nuevo){
        if (total_utilizados < TAMANIO){
            vector_privado[total_utilizados] = nuevo;
            total_utilizados++;
        }
    }
    double Elemento(int indice){
        return vector_privado[indice];
    }
};
```

Los lenguajes de programación ofrecen recursos para no tener que escribir el mismo código para **tipos distintos** :

- ▷ *Plantillas (Templates)* en C++
- ▷ *Genéricos (Generics)* en Java y .NET

Por ahora, tendremos que duplicar el código y crear una clase para cada tipo de dato. 😞

Veamos cómo quedarían implementados los métodos de búsqueda vistos en la sección **III.2.1** dentro de la clase `SecuenciaCaracteres`:

IV.2.9.2. Métodos de búsqueda

Búsqueda secuencial

¿Devolvemos un `bool`? Mejor si devolvemos la posición en la que se ha encontrado. En caso contrario, devolvemos un valor imposible de posición (por ejemplo, -1)

Cabecera:

```
class SecuenciaCaracteres{
    .....
    int PrimeraOcurrencia (char buscado){ ..... }
};
```

Llamada:

```
int main(){
    SecuenciaCaracteres secuencia;
    .....
    pos_encontrado = secuencia.PrimeraOcurrencia('l');

    if (pos_encontrado == -1)
        cout << "\nNo encontrado";
    else
        cout << "\nEncontrado en la posición " << pos_encontrado;
}
```

Implementación:

```
class SecuenciaCaracteres{
private:
    static const int TAMANIO = 50;
    char vector_privado[TAMANIO];
    int total_utilizados;
public:
    .....
    int PrimeraOcurrencia (char buscado){
        bool encontrado = false;
        int i = 0;

        while (i < total_utilizados && !encontrado){
            if (vector_privado[i] == buscado)
                encontrado = true;
            else
                i++;
        }

        if (encontrado)
            return i;
        else
            return -1;
    }
};
```


Observe que también podríamos haber puesto lo siguiente:

```
int PrimeraOcurrencia (char buscado){  
    .....  
    while (i < total_utilizados  &&  !encontrado)  
        if (Elemento(i) == buscado)  
            ....  
}  
};
```

- ▷ La llamada al método `Elemento` es más *segura* ya que dentro de él podemos comprobar la condición de que el índice sea correcto.
- ▷ Por otra parte, la llamada conlleva una recarga computacional. Es más rápido acceder directamente a la componente del vector privado. No sería significativa si sólo se realizase una llamada, pero está dentro de un bucle, por lo que la recarga sí puede ser importante.

Por tanto, normalmente optaremos por trabajar dentro de los métodos de la clase accediendo directamente al vector privado con la notación corchete.

Consejo: *Procure implementar los métodos de una clase lo más eficientemente posible, pero sin que ello afecte a ningún cambio en la interfaz de ésta (las cabeceras de los métodos públicos)*



A veces, podemos estar interesados en buscar entre una componente izquierda y otra derecha, ambas inclusive.

```
class SecuenciaCaracteres{
private:
    static const int TAMANIO = 50;
    char vector_privado[TAMANIO];
    int total_utilizados;
public:
    .....
    int PrimeraOcurrenciaEntre (int pos_izda, int pos_dcha, char buscado){
        int i = pos_izda;
        bool encontrado = false;

        while (i <= pos_dcha && !encontrado)
            if (vector_privado[i] == buscado)
                encontrado = true;
            else
                i++;

        if (encontrado)
            return i;
        else
            return -1;
    }
};
```

La clase puede proporcionar los dos métodos `PrimeraOcurrencia` y `PrimeraOcurrenciaEntre`. Pero para no repetir código, cambiaríamos la implementación del primero como sigue:

```
int PrimeraOcurrencia (char buscado){
    return PrimeraOcurrenciaEntre (0, total_utilizados - 1, buscado);
}
```

Hasta ahora, nos quedaría:

SecuenciaCaracteres	
- const int	<u>TAMANIO</u>
- char	vector_privado[TAMANIO]
- int	total_utilizados
+	SecuenciaCaracteres()
+ int	TotalUtilizados()
+ int	Capacidad()
+ void	Aniade(char nuevo)
+ void	Modifica(int pos_a_modificar, char valor_nuevo)
+ char	Elemento(int indice)
+ void	EliminaTodos()
+ int	PrimeraOcurrenciaEntre(int pos_izda, int pos_dcha, char buscado)
+ int	PrimeraOcurrencia(char buscado)

Ampliación:



Observe que el cliente (el que use) de la clase `SecuenciaCaracteres` debe saber que `-1` es el valor que indica que la posición no existe. Para evitarlo, la clase puede proporcionar dicha información a través de una constante estática pública:

```
class SecuenciaCaracteres{
.....
public:
    static const int POS_NO_EXISTE = -1;

    int PrimeraOcurrencia(char buscado){
        .....
        if (encontrado)
            return i;
        else
            return POS_NO_EXISTE;
    }
};

int main(){
    .....
    if (secuencia.PrimerOcurrencia(a_buscar)
        == secuencia.POS_NO_EXISTE){
        .....
    }
```

Las constantes estáticas también pueden referenciarse sin usar ningún objeto; basta el nombre de la clase. Por lo tanto, el siguiente código también sería válido:

```
if (secuencia.PrimerOcurrencia(a_buscar)
    == SecuenciaCaracteres.POS_NO_EXISTE){
    .....
}
```

Búsqueda binaria

Precondiciones de uso: Se aplica sobre un vector ordenado.

No comprobamos que se cumple la precondición dentro del método ya que resultaría muy costoso comprobar que el vector está ordenado cada vez que ejecutamos la búsqueda.

La cabecera del método no cambia, aunque para destacar que no es una búsqueda cualquiera, sino que necesita una precondición, cambiamos el nombre del identificador.

Cabecera:

```
class SecuenciaCaracteres{
    .....
    // Prec: secuencia ya ordenada
    int BusquedaBinaria (char buscado){ ..... }
};
```

Llamada:

```
int main(){
    SecuenciaCaracteres secuencia;
    .....
    pos_encontrado = secuencia.BusquedaBinaria('B');

    if (pos_encontrado == -1)
        cout << "\nNo encontrado";
    else
        cout << "\nEncontrado en la posición " << pos_encontrado;
}
```

Implementación:

```
class SecuenciaCaracteres{
private:
    static const int TAMANIO = 50;
    char vector_privado[TAMANIO];
    int total_utilizados;
public:
    .....
    int BusquedaBinaria (char buscado){
        int izda, dcha, centro;
        bool encontrado = false;

        izda = 0;
        dcha = total_utilizados - 1;

        while (izda <= dcha && !encontrado){
            centro = (izda + dcha) / 2;

            if (vector_privado[centro] == buscado)
                encontrado = true;
            else if (buscado < vector_privado[centro])
                dcha = centro - 1;
            else
                izda = centro + 1;
        }

        if (encontrado)
            return centro;
        else
            return -1;
    } };
```

Buscar el mínimo

Queremos calcular el valor mínimo de una secuencia ¿Devolvemos el valor en sí? Mejor si devolvemos su posición por si luego quisiéramos acceder a la componente.

Como norma general, cuando un método busque algo en un vector, debe devolver la posición en la que se encuentra y no el elemento en sí.

Cabecera:

```
class SecuenciaCaracteres{
    .....
    int PosicionMinimo() { ..... }
};
```

Llamada:

```
int main(){
    SecuenciaCaracteres secuencia;
    .....
    pos_minimo = secuencia.PosicionMinimo();
}
```

Implementación:

```
class SecuenciaCaracteres{
    .....
    int PosicionMinimo(){
        int posicion_minimo;
        char minimo;

        if (total_utilizados > 0){
            minimo = vector_privado[0];
            posicion_minimo = 0;

            for (int i = 1; i < total_utilizados ; i++){
                if (vector_privado[i] < minimo){
                    minimo = vector_privado[i];
                    posicion_minimo = i;
                }
            }
        }
        else
            posicion_minimo = -1;

        return posicion_minimo;
    }
};
```


Buscar el mínimo en una zona del vector

Debemos pasar como parámetro al método los índices `izda` y `dcha` que delimitan la zona en la que se quiere buscar.

```
v = (h,b,t,c,f,i,d,f,?,?,?)   izda = 2 , dcha = 5
```

`PosicionMinimoEntre(2, 5)` devolvería la posición 3 (índice del vector)

Cabecera:

```
class SecuenciaCaracteres{
    .....
    // Precond: 0 <= izda <= dcha < total_utilizados
    int PosicionMinimoEntre(int izda, int dcha){ ..... }
};
```

Llamada:

```
int main(){
    SecuenciaCaracteres secuencia;
    .....
    pos_minimo = secuencia.PosicionMinimoEntre(1,3);
}
```

Implementación:

```
class SecuenciaCaracteres{
private:
    static const int TAMANIO = 50;
    char vector_privado[TAMANIO];
    int total_utilizados;
public:
    .....
    // Precond: 0 <= izda <= dcha < total_utilizados
    int PosicionMinimoEntre(int izda, int dcha){
        int posicion_minimo = -1;
        char minimo;

        minimo = vector_privado[izda];
        posicion_minimo = izda;

        for (int i = izda+1 ; i <= dcha ; i++)
            if (vector_privado[i] < minimo){
                minimo = vector_privado[i];
                posicion_minimo = i;
            }

        return posicion_minimo;
    }
};
```

Al igual que hicimos con el método `PrimeraOcurrencia`, la implementación de `PosicionMinimo` habría que cambiarla por:

```
int PosicionMinimo(){
    return PosicionMinimoEntre(0, total_utilizados - 1);
}
```

Nos quedaría por ahora:

SecuenciaCaracteres	
- const int	<u>TAMANIO</u>
- char	vector_privado[TAMANIO]
- int	total_utilizados
+	SecuenciaCaracteres()
+ int	TotalUtilizados()
+ int	Capacidad()
+ void	Aniade(char nuevo)
+ void	Modifica(int pos_a_modificar, char valor_nuevo)
+ char	Elemento(int indice)
+ void	EliminaTodos()
+ int	PrimeraOcurrenciaEntre(int pos_izda, int pos_dcha, char buscado)
+ int	PrimeraOcurrencia(char buscado)
+ int	BusquedaBinaria(char buscado)
+ int	PosicionMinimo()
+ int	PosicionMinimoEntre(int izda, int dcha)

http://decsai.ugr.es/jccubero/FP/IV_secuencia_busqueda.cpp

Inserción de un valor

La inserción de un valor dentro del vector implica desplazar las componentes que hay a su derecha. Por lo tanto, decidimos comprobar que los parámetros son correctos para evitar modificar componentes no reservadas (tal y como recomendábamos en la página 542).

Cabecera:

```
class SecuenciaCaracteres{
    .....
    void Inserta(int pos_insercion, char valor_nuevo){ ..... }
};
```

Llamada:

```
int main(){
    SecuenciaCaracteres secuencia;
    .....
    secuencia.Inserta(2, 'r');
```

Implementación:

```
class SecuenciaCaracteres{
private:
    static const int TAMANIO = 50;
    char vector_privado[TAMANIO];
    int total_utilizados;
public:
    .....
    void Inserta(int pos_insercion, char valor_nuevo){
        if (total_utilizados < TAMANIO && pos_insercion >= 0
            && pos_insercion <= total_utilizados){

            for (int i = total_utilizados ; i > pos_insercion ; i--)
                vector_privado[i] = vector_privado[i-1];

            vector_privado[pos_insercion] = valor_nuevo;
            total_utilizados++;
        }
    }
}
```

Eliminación de un valor

Debemos pasar como parámetro la posición a eliminar. De nuevo, para evitar modificar posiciones no reservadas, decidimos comprobar que los parámetros son correctos (ver página 542)

Cabecera:

```
class SecuenciaCaracteres{
    .....
    void Elimina(int pos_a_eliminar){
        .....
    }
};
```

Llamada:

```
int main(){
    SecuenciaCaracteres secuencia;
    .....
    secuencia.Elimina(2);
}
```

Implementación:

```
class SecuenciaCaracteres{
private:
    static const int TAMANIO = 50;
    char vector_privado[TAMANIO];
    int total_utilizados;
public:
    .....
    // Elimina una componente, dada por su posición
    void Elimina (int posicion){
        if (posicion >= 0 && posicion < total_utilizados){
            int tope = total_utilizados-1;

            for (int i = posicion ; i < tope ; i++)
                vector_privado[i] = vector_privado[i+1];

            total_utilizados--;
        }
    }
};
```

Nos quedaría por ahora:

SecuenciaCaracteres	
- const int	<u>TAMANIO</u>
- char	vector_privado[TAMANIO]
- int	total_utilizados
+	SecuenciaCaracteres()
+ int	TotalUtilizados()
+ int	Capacidad()
+ void	Aniade(char nuevo)
+ void	Modifica(int pos_a_modificar, char valor_nuevo)
+ char	Elemento(int indice)
+ void	EliminaTodos()
+ int	PrimeraOcurrenciaEntre(int pos_izda, int pos_dcha, char buscado)
+ int	PrimeraOcurrencia(char buscado)
+ int	BusquedaBinaria(char buscado)
+ int	PosicionMinimoEntre(int izda, int dcha)
+ int	PosicionMinimo()
+ void	Inserta(int pos_insercion, char valor_nuevo)
+ void	Elimina(int pos_a_eliminar)

http://decsai.ugr.es/jccubero/FP/IV_secuencia_inserta_elimina.cpp

IV.2.9.3. Algoritmos de ordenación

Se usarán para ordenar los caracteres según el orden de la tabla ASCII. Obviamente, sólo tendrá sentido en ciertas situaciones. Por ejemplo, si representamos las notas según una escala ECTS (con letras) podríamos estar interesados en ordenar dichas notas. En cualquier caso, es cierto que habrá pocos problemas que necesiten ordenar los caracteres de una secuencia. Caso distinto sería que los datos de tipo `char` se utilizasen para representar un entero pequeño, como por ejemplo, los datos de un sensor. En esta situación, sí podría tener sentido ordenar una secuencia de `char`.

Cabecera:

```
class SecuenciaCaracteres{  
    .....  
    void Ordena(){ ..... }  
};
```

Llamada:

```
int main(){  
    SecuenciaCaracteres secuencia;  
    .....  
    secuencia.Ordena();  
}
```

Para diferenciar los distintos algoritmos de ordenación usaremos nombres de métodos distintos.

Es importante que observe el uso de otros métodos de la clase, para implementar los distintos algoritmos de ordenación.

Ordenación por selección

Implementación:

```
class SecuenciaCaracteres{
private:
    static const int TAMANIO = 50;
    char vector_privado[TAMANIO];
    int total_utilizados;

    void IntercambiaComponentes_en_Posiciones(int pos_izda, int pos_dcha){
        char intercambia;

        intercambia = vector_privado[pos_izda];
        vector_privado[pos_izda] = vector_privado[pos_dcha];
        vector_privado[pos_dcha] = intercambia;
    }
public:
    int PosicionMinimoEntre(int izda, int dcha){
        .....
    }
    .....
    void Ordena_por_Seleccion(){
        int pos_min;

        for (int izda = 0 ; izda < total_utilizados ; izda++){
            pos_min = PosicionMinimoEntre(izda, total_utilizados-1);
            IntercambiaComponentes_en_Posiciones(izda, pos_min);
        }
    }
    .....
}
```

Observe que no hemos comprobado las precondiciones en el método privado `IntercambiaComponentes_en_Posiciones`. Esto lo hemos hecho para aumentar la eficiencia de los métodos públicos que lo usen (por ejemplo, los métodos de ordenación) Si quisiéramos ofrecer un método público que hiciese lo mismo, sí habría que comprobar las precondiciones.

*Para aumentar la eficiencia, a veces será conveniente omitir la comprobación de las precondiciones en los métodos **privados**.*

Observe la diferencia de este consejo sobre los métodos privados con el que vimos en la página 581 sobre los métodos públicos.

Ordenación por inserción

Implementación:

```
class SecuenciaCaracteres{
private:
    static const int TAMANIO = 50;
    char vector_privado[TAMANIO];
    int total_utilizados;
public:
    .....
    void Ordena_por_Insercion(){
        int izda, i;
        char a_desplazar;

        for (izda = 1; izda < total_utilizados; izda++){
            a_desplazar = vector_privado[izda];

            for (i = izda; i > 0 && a_desplazar < vector_privado[i-1]; i--){
                vector_privado[i] = vector_privado[i-1];

                vector_privado[i] = a_desplazar;
            }
        }
    };
};
```

Ordenación por burbuja

► *Primera Aproximación*

```
class SecuenciaCaracteres{
private:
    static const int TAMANIO = 50;
    char vector_privado[TAMANIO];
    int total_utilizados;

    void IntercambiaComponentes_en_Posiciones(int pos_izda, int pos_dcha){
        char intercambia;

        intercambia = vector_privado[pos_izda];
        vector_privado[pos_izda] = vector_privado[pos_dcha];
        vector_privado[pos_dcha] = intercambia;
    }
public:
    .....
    void Ordena_por_Burbuja(){
        int izda, i;

        for (izda = 0; izda < total_utilizados; izda++)
            for (i = total_utilizados-1 ; i > izda ; i--)
                if (vector_privado[i] < vector_privado[i-1])
                    IntercambiaComponentes_en_Posiciones(i, i-1);
    }
};
```

► Segunda Aproximación

```
class SecuenciaCaracteres{
private:
    static const int TAMANIO = 50;
    char vector_privado[TAMANIO];
    int total_utilizados;

    void IntercambiaComponentes_en_Posiciones(int pos_izda, int pos_dcha){
        .....
    }
public:
    .....
    void Ordena_por_BurbujaMejorado(){
        int izda, i;
        bool cambio;

        cambio = true;

        for (izda = 0; izda < total_utilizados && cambio; izda++){
            cambio = false;

            for (i = total_utilizados-1; i > izda; i--){
                if (vector_privado[i] < vector_privado[i-1]){
                    IntercambiaComponentes_en_Posiciones(i, i-1);
                    cambio = true;
                }
            }
        }
    };
};
```

Nos quedaría por ahora:

SecuenciaCaracteres	
- const int	<u>TAMANIO</u>
- char	vector_privado[TAMANIO]
- int	total_utilizados
- void	IntercambiaComponentes_en_Posiciones(int pos_izda, int pos_dcha)
+	SecuenciaCaracteres()
+ int	TotalUtilizados()
+ int	Capacidad()
+ void	Aniade(char nuevo)
+ void	Modifica(int pos_a_modificar, char valor_nuevo)
+ char	Elemento(int indice)
+ void	EliminaTodos()
+ int	PrimeraOcurrenciaEntre(int pos_izda, int pos_dcha, char buscado)
+ int	PrimeraOcurrencia(char buscado)
+ int	BusquedaBinaria(char buscado)
+ int	PosicionMinimoEntre(int izda, int dcha)
+ int	PosicionMinimo()
+ void	Inserta(int pos_insercion, char valor_nuevo)
+ void	Elimina(int pos_a_eliminar)
+ void	Ordena_por_Seleccion()
+ void	Ordena_por_Insercion()
+ void	Ordena_por_Burbuja()
+ void	Ordena_por_BurbujaMejorado()

http://decsai.ugr.es/jccubero/FP/IV_secuencia_ordenacion.cpp

Nota:

Normalmente, las clases no tendrán un número elevado de métodos. La clase **SecuenciaCaracteres** sería una excepción. Este tipo de clases, contenedoras de datos de un mismo tipo, son usuales en muchos lenguajes de programación (por ejemplo, la clase **ArrayList** en la plataforma **.NET**)

IV.2.10. La clase string

IV.2.10.1. Métodos básicos

Ya vimos en la página 329 que el tipo `string` usado hasta ahora es, realmente, una clase por lo que los datos de tipo `string` son objetos.

Podemos pensar que, internamente, trabaja con un dato miembro privado que es un vector de `char`. Así es como hemos definido la clase `SecuenciaCaracteres`. La diferencia está en que el tipo de vector usado en la clase `string` tiene una tamaño variable. En el segundo cuatrimestre aprenderá a trabajar con secuencias de datos de tamaño variable.

string	
-
+ void	push_back(char nuevo)
+ size_type	capacity()
+ size_type	size()
+ void	clear()
+ reference	at(size_type indice)

Los métodos `push_back` y `size` ya los conocemos. Sin entrar en detalles:

- ▷ El método `capacity` devuelve la capacidad máxima actual (puede variar) y el método `clear` la deja con "".
- ▷ `size_type` es un tipo entero que garantiza albergar el tamaño necesario de un `string`.
- ▷ `reference` es una referencia (se verá en el segundo cuatrimestre). Por ahora, puede usarlo pensando que devuelve un `char` del `string`.

La clase `SecuenciaCaracteres` **se ha definido con métodos similares a los existentes en un** `string`:

<code>string</code>	<code>SecuenciaCaracteres</code>
<code>push_back</code>	<code>Aniade</code>
<code>capacity</code>	<code>Capacidad</code>
<code>size</code>	<code>TotalUtilizados</code>
<code>clear</code>	<code>EliminaTodos</code>
<code>at</code>	<code>Elemento</code>

Ampliación:



Otros métodos interesantes (no hay que aprenderlos y no entran en el examen):

at. Accede (y modifica) a las componentes de forma segura.

append. Añade una cadena al final de otra.

insert. Inserta una subcadena a partir de una posición.

replace. Reemplaza una serie de caracteres consecutivos por una subcadena.

erase. Borra un número de caracteres, a partir de una posición.

find. Encuentra un carácter o una subcadena.

substr. Obtiene una subcadena.

reserve. Reserva un número de componentes (número que luego puede aumentar o disminuir). Útil cuando deseemos trabajar con un `string` grande, para que así, el programa no tenga que estar aumentando el tamaño del vector interno conforme se va llenando (lo que puede ser muy costoso).

La clase `string` permite además operaciones más *sofisticadas* como:

- ▷ Asignar un valor concreto al objeto `string` a través de un literal de cadena de caracteres.
- ▷ Imprimir su contenido con `cout`.
- ▷ Acceder a las componentes individuales como si fuese un vector clásico, con la notación corchete.
- ▷ Concatenar dos cadenas o una cadena con un carácter usando el operador `+`
- ▷ Al crear un `string` por primera vez, éste contiene la *cadena vacía* (*empty string*), a saber `""`

```
#include <iostream>
#include <string>
using namespace std;

int main(){
    string cadena;

    if (cadena == "")
        cout << "Vacía";           // Vacía

    cadena.push_back('F');
    cout << "\n" << cadena;        // F

    cadena = "Funda";              // Funda
    cadena = cadena + "ment";       // Fundament
    cadena = cadena + 'o';          // Fundamento
    cout << "\n" << cadena;        // Fundamento
    cadena.push_back('s');
    cout << "\n" << cadena;        // Fundamentos
    cout << "\n" << cadena.capacity(); // 20
    cout << "\n" << cadena.size();  // 11
    cadena[9] = 'a';
    cout << "\n" << cadena;        // Fundamentas
    cadena[12] = 'z';               // Comportamiento indeterminado
```



IV.2.10.2. El método ToString

En numerosas ocasiones, será bastante útil proporcionar un método ToString a las clases que construyamos. Ya lo hicimos con la clase Fecha, por ejemplo (ver página 485) Para una clase CalificacionesFinales podría devolver una cadena con los nombres y notas de todos los alumnos.

En la clase SecuenciaCaracteres, está clara su utilidad:

```
class SecuenciaCaracteres{
.....
    string ToString(){
        // Si el número de caracteres en memoria es muy grande,
        // es mucho más eficiente reservar memoria previamente
        // y usar push_back

        string cadena;

        cadena.reserve(total_utilizados);

        for (int i=0; i < total_utilizados; i++)
            cadena.push_back(vector_privado[i]);
        // cadena = cadena + vector_privado[i]  <- Evitarlo.
        // Muy muy ineficiente con muchas iteraciones;

        return cadena;
    }
};

int main(){
    SecuenciaCaracteres secuencia;
    .....
    cout << "\n" << secuencia.ToString();
}
```

IV.2.10.3. Lectura de un string con getline

¿Cómo leemos una variable de tipo `string`? `cin` se salta los separadores:

```
string cadena1, cadena2;

cout << "\nIntroduzca nombre: "; // "Juan    Carlos"
cin >> cadena1;
cin >> cadena2;
cout << "\n\n" << "-" << cadena1 << "-" << cadena2 << "-";
                        // -Juan-Carlos-
```

Si queremos leer los separadores dentro de un `string` usaremos la siguiente función que usa *getline* (no hace falta entender cómo lo hace)

```
string LeeCadenaHastaEnter(){
    string cadena;
    // Nos saltamos todos los \n que pudiera haber al principio:

    do{
        getline(std::cin, cadena);
    }while (cadena == "");

    return cadena;
}

int main(){
    string cadena1, cadena2;

    cout << "\nIntroduzca nombre: ";
    cadena1 = LeeCadenaHastaEnter(); // Juan Carlos Cubero
    cout << "\nIntroduzca nombre: "; // Pedro Jiménez
    cadena2 = LeeCadenaHastaEnter();
    cout << "\n\n" << "-" << cadena1 << "-" << cadena2 << "-";
                        // -Juan Carlos Cubero-Pedro Jiménez-
```

IV.2.11. Diseño de una clase

El diseño de una solución a un problema, basada en programación dirigida a objetos, es un tema complejo que es imposible abarcar en una única asignatura. Deben dominarse conceptos y técnicas como Herencia, Polimorfismo, Patrones de Diseño, Modelos de Representación de Conocimiento, Metodologías de desarrollo, etc. En este apartado vamos a ver algunos principios y consejos que debemos tener en cuenta a la hora de diseñar una clase.

IV.2.11.1. Datos miembro y parámetros

En la página 494 vimos que debíamos limitar los datos miembro a aquellos datos que eran compartidos por una gran parte de los métodos de la clase. Ahora bien, no siempre es fácil determinar cuáles deben ser los datos miembros y cuáles los parámetros. Una heurística de ayuda es la siguiente:

Cuando no tengamos claro si un dato ha de ser dato miembro o parámetro de un método: si suele cambiar continuamente durante la vida del objeto, es candidato a ser parámetro. Si no cambia o lo hace poco, es candidato a ser dato miembro.

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.

Ejemplo. Recuperamos el ejemplo del cómputo del salario de la página 198.

Supongamos que gestionamos varias sucursales de la misma empresa y que los criterios de subida (experiencia, edad, número de hijos, etc) son los mismos, salvo que los límites pueden variar de una a otra. En cualquier caso, una vez establecidos ya no cambiarán. Supongamos también que el salario base no suele cambiar mucho dentro de la misma empresa.

Con una función tendríamos:

```
double SalarioFinal(int minimo_experiencia_alta,
                    int minimo_familia_numerosa,
                    int minimo_edad_senior,
                    double maximo_salario_bajo,
                    double salario_base,
                    int experiencia,
                    int edad,
                    int numero_hijos)
```

Queda mucho mejor una clase en la que:

- ▷ Los datos que varían poco durante la vida del objeto serán datos miembro y los establecemos en el constructor. Por ejemplo, los límites para establecer el salario final.
- ▷ Los datos que van variando constantemente durante la vida del objeto serán parámetros a los métodos. Por ejemplo, la experiencia, edad y número de hijos de cada empleado.
- ▷ Habrá datos que puedan variar algo y que son necesarios para los métodos de la clase. Éstos serán datos miembro que se establecerán con métodos o en el constructor (preferible).

CalculadoraSalario	
- int	minimo_experiencia_alta
- int	minimo_familia_numerosa
- int	minimo_edad_senior
- double	maximo_salario_bajo
- double	salario_base
- bool	EsCorrectoSalario(double un_salario)
+	CalculadoraSalario(int minimo_experiencia_alta_sucursal, int minimo_familia_numerosa_sucursal, int minimo_edad_senior_sucursal, int maximo_salario_bajo_sucursal)
+ void	SetSalarioBase(double salario_base_mensual_trabajador)
+ double	SalarioFinal(int experiencia, int edad, int numero_hijos)

```
.....
int main(){
    // Orden de los parámetros en el constructor:
    // mínimo experiencia alta, mínimo familia numerosa,
    // mínimo edad senior, máximo salario bajo

    CalculadoraSalario calculadora_salario_Jaen(2, 2, 45, 1300);
    CalculadoraSalario calculadora_salario_Granada(3, 3, 47, 1400);
    .....
    calculadora_salario_Jaen.SetSalarioBase(salario_base_Jaen);
    .....
    // Bucle de lectura de datos:
    do{
        .....
        cin >> experiencia;
        cin >> edad;
        cin >> numero_hijos;
        .....
        salario_final =
            calculadora_salario_Jaen.SalarioFinal(experiencia,
                                                    edad,
                                                    numero_hijos);

        .....
    }while (hay_datos_por_procesar);
    .....
}
```

http://decsai.ugr.es/jccubero/FP/IV_actualizacion_salarial.cpp

IV.2.11.2. 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.**

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
<hr/>	
- string	nombre
- string	dni
- int	dia_ncmto
- int	mes_ncmto
- int	anio_ncmto
- void	SetIdentificador(string identificador_cuenta)
+	ClienteCuentaBancaria(double saldo_inicial, string nombre_cliente, string dni_cliente, int dia_nacimiento, int mes_nacimiento, int anio_nacimiento)
+	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.

IV.2.11.3. Separación de Entradas/Salidas y Cálculos

En el tema II (ver página 201) se vio la necesidad de separar los bloques de Entrada y Salida de datos de los bloques de cálculos. ¿Cómo se traduce esta necesidad al trabajar con clases? ¿Y qué relación tiene con el principio de responsabilidad única?

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 responsabilidad única. 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.

Una consecuencia del Principio de Responsabilidad Única es que jamás mezclaremos en una misma clase métodos de *cálculo* con métodos que accedan a los dispositivos de entrada/salida.

Así pues, tendremos clases que se encargarán únicamente de realizar las E/S de nuestro programa y otras clases para realizar el resto de cálculos.



Ejemplo. Retomamos el ejemplo de la fecha:

¿Nos preguntamos si hubiera sido más cómodo sustituir el método ToString por Imprimir?:

```
class Fecha {
private:
    int dia, mes, anio;
    .....
public:
    .....
    void Imprime(){
        cout << string(dia) + "/" + to_string(mes)
            + "/" + to_string(anio);
    }
};

int main(){
    Fecha nacimiento_JC (27,2,1987);
    nacimiento_JC.Imprime();
}
```



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 to_string(dia) + "/" + to_string(mes)
            + "/" + to_string(anio);
    }
};

int main(){
    Fecha nacimiento_JC (27,2,1987);

    cout << nacimiento_JC.ToString();
}
```



IV.2.11.4. Tareas primitivas

Para resolver una tarea medianamente compleja, tendremos que descomponerla en subtareas de más fácil resolución. Normalmente, cada una de ellas será resuelta con un método. Intentaremos que sean métodos los más reutilizables posible.

Ejemplo. Sobre la clase `SecuenciaCaracteres` queremos borrar el elemento mínimo de la secuencia.

¿Construimos el siguiente método?

```
void EliminaMinimo()
```

Lo normal será que no lo hagamos ya que es un método demasiado específico y no es de esperar que sea muy utilizado por los clientes de la clase. Mucho mejor si definimos un método más genérico y con más posibilidades de reutilizar en el futuro como es el siguiente:

```
void Elimina (int posicion_a_eliminar)
```

Desde fuera de la clase, si queremos eliminar el mínimo de la secuencia, basta realizar la siguiente llamada:

```
int main(){  
    SecuenciaCaracteres cadena;  
    .....  
    cadena.Elimina( cadena.PosicionMinimo() );  
}
```

En cualquier caso, si se prevé que la eliminación del mínimo va a ser una operación muy usual, puede añadirse como un método de la clase, pero eso sí, en la implementación se llamará a los otros métodos.

```
class SecuenciaCaracteres{
    .....
    void EliminaMinimo(){
        Elimina( PosicionMinimo() );
    }
};

int main(){
    SecuenciaCaracteres cadena;
    .....
    cadena.EliminaMinimo();
}
```

A la hora de diseñar la interfaz de una clase (el conjunto de métodos públicos), intente construir un conjunto minimal de operaciones primitivas que puedan reutilizarse lo máximo posible.

IMPORTANT

IV.2.11.5. 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 507. 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,
           y_1,
           x_2,
           y_2;
    bool SonCorrectas(double abs_1, double ord_1,
                      double abs_2, double 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 la función SonIguales
    .....
};

```

Ejemplo. Definamos la clase `Fraccion` para representar una fracción matemática. Los valores de numerador y denominador los pasamos como parámetros al constructor y no permitimos su posterior modificación.

Fraccion	
- int	numerador
- int	denominador
+ Fraccion(int el_numerador, int el_denominador)	
+ int	Numerador()
+ int	Denominador()
+ void	Reduce()
+ double	Division()
+ string	ToString()

```
int main(){
    Fraccion una_fraccion(4, 10);    // 4/10

    una_fraccion.Reduce();            // 2/5
    cout << una_fraccion.ToString();
    .....
}
```

Para implementar el método `Reduce`, 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 Reduce(){
        int mcd;

        mcd = MCD(numerador, denominador);
        numerador = numerador/mcd;
        denominador = denominador/mcd;
    }
};

int main(){
    Fraccion una_fraccion(4, 10);    // 4/10

    una_fraccion.Reduce();           // 2/5
    cout << una_fraccion.ToString();
    .....
}
```

http://decsai.ugr.es/jccubero/FP/IV_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.

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.**

Resúmenes:

Los parámetros formales son los datos que la función necesita conocer del exterior, para poder hacer sus cálculos

El valor calculado por la función es el resultado de evaluar la expresión en la sentencia `return`

Una función debería devolver siempre un valor. En caso contrario, se produce un **comportamiento indeterminado** (*undefined behaviour*) (recuerde lo visto en la página 318).

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.

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)

Cuando construya las funciones que resuelven tareas concretas de un programa, procure diseñarlas de tal forma que se favorezca su reutilización en otros programas.

No todas las funciones se reutilizarán en otros contextos: habrá que encontrar un equilibrio entre funciones más genéricas y funciones más específicas de una solución concreta.

Todos los datos auxiliares que la función necesite para realizar sus cálculos serán datos locales.

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.

Los parámetros nos permiten aumentar la flexibilidad en el uso de la función.

Al diseñar la cabecera de una función, el programador debe analizar detalladamente cuáles son los factores que influyen en la tarea que ésta resuelve y así determinar los parámetros apropiados.

Una clase es un tipo de dato.

Los objetos son datos cuyo tipo de dato es una clase.

La encapsulación es el mecanismo de modularización utilizado en PDO. La idea consiste en aunar datos y comportamiento en un mismo módulo.

Cada vez que modificamos un dato miembro, diremos que se ha modificado el estado del objeto.

Las inicializaciones especificadas en la declaración de los datos miembro dentro de la clase son posibles desde C++11. Éstas se aplican en el momento que se crea un objeto cualquiera de dicha clase.

Los métodos acceden a los datos miembro directamente.

Los métodos pueden modificar el estado del objeto.

Usaremos nombres para denotar las clases y verbos para los métodos de tipo `void`. Normalmente no usaremos infinitivos sino la tercera persona del singular.

Para los métodos que devuelven un valor, usaremos nombres (el del valor que devuelve).

Dentro de la clase, los métodos pueden llamarse unos a otros. Esto nos permite cumplir el principio de una única vez.

Los métodos permiten establecer la política de acceso a los datos miembro, es decir, determinar cuáles son las operaciones permitidas con ellos.

Será usual que, dentro de los métodos `Set`, comprobemos que los parámetros pasados sean correctos. Si no lo son, lo normal será que no hagamos nada dentro del método y mantengamos los que hubiese.

En cualquier caso, también es lícito no realizar ninguna comprobación e imponer la correspondiente restricción como precondition del método `Set`.

Los métodos privados se usan para realizar tareas propias de la clase que no queremos que se puedan invocar desde fuera de ésta.

Los constructores nos permiten ejecutar automáticamente un conjunto de instrucciones, cada vez que se crea un objeto.

Si el programador define cualquier constructor (con o sin parámetros) ya no está disponible el constructor de oficio.

Dentro de un constructor podemos ejecutar código y llamar a métodos de la clase.

Para asignar a los datos miembro un valor por defecto lo podemos hacer de dos formas:

- ▷ ***O bien usamos la inicialización de C++ 11 en la declaración de los datos miembro.***
- ▷ ***O bien lo asignamos directamente en el constructor***

Podemos definir varios constructores. Esto nos permite crear objetos de maneras distintas, según nos convenga.

Si se desea un constructor sin parámetros, habrá que crearlo explícitamente, junto con los otros constructores.

Debemos evitar, en la medida de lo posible, que un objeto esté en un estado inválido (zombie) en algún momento de la ejecución del programa. Para ello:

▷ ***Cuando el objeto se construya por primera vez, podemos asignar valores por defecto específicos a los datos miembro. Esto lo haremos sólo cuando tenga sentido: saldo a 0 por defecto en una cuenta bancaria, circunferencia goniométrica por defecto, etc.***

▷ ***Si no tiene sentido usar valores por defecto, podemos definir un constructor y pasarle como parámetros actuales los valores de los datos miembro.***

Para simplificar el código de las clases, a lo largo de la asignatura asumiremos que dichos parámetros actuales serán correctos

▷ ***En cualquier caso, con las herramientas vistas hasta ahora, no siempre será posible resolver este problema y a veces tendremos que aceptar que un objeto pueda estar temporalmente en un estado inválido.***

¿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)

*Si decidimos que es necesario **comprobar** alguna precondición dentro de un método/función y ésta se viola, ¿qué debemos hacer como respuesta?*

- ▷ *O bien no hacemos nada 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. La notificación jamás se hará imprimiendo un mensaje en pantalla desde dentro del método/función.*
- ▷ *En el tema V se verá un apartado de Ampliación en el que se muestra cómo resolver apropiadamente los dos problemas anteriores usando el mecanismo de las excepciones.*

Las constantes a nivel de objeto presentan algunas dificultades (ver Tema V) Por lo tanto, durante la asignatura, fomentaremos el uso de constantes estáticas en detrimento de las constantes a nivel de objeto.

*El valor de las constantes estáticas se asignan durante la compilación. Por tanto, está permitido usar datos miembro constantes **públicos***

Las funciones o métodos pueden definir vectores como datos locales para hacer sus cálculos. Pero no pueden devolverlos.

Lo que sí podremos hacer es devolver objetos que contengan vectores.

En esta asignatura no pasaremos los vectores como parámetros de las funciones o métodos.

Como norma general, cuando un método busque algo en un vector, debe devolver la posición en la que se encuentra y no el elemento en sí.

Para aumentar la eficiencia, a veces será conveniente omitir la comprobación de las precondiciones en los métodos **privados**.

Cuando no tengamos claro si un dato ha de ser dato miembro o parámetro de un método: si suele cambiar continuamente durante la vida del objeto, es candidato a ser parámetro. Si no cambia o lo hace poco, es candidato a ser dato miembro.

Por normal general, cuantos más métodos de una clase utilicen todos los datos miembros de la misma, mayor será la cohesión de la clase.

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.

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.*



Consejo: *Salvo en el caso de funciones con muy pocas líneas de código, evite la inclusión de sentencias `return` perdidas dentro del código de la función. Use mejor un único `return` al final de la función.*



Consejo: *No abuse de la sobrecarga de funciones que únicamente difieran en los tipos de los parámetros formales*



Consejo: *En la medida de lo posible, evite que durante la ejecución del programa, existan objetos en un estado inválido (zombies) en memoria*



Consejo: *Procure implementar los métodos de una clase lo más eficientemente posible, pero sin que ello afecte a ningún cambio en la interfaz de ésta (las cabeceras de los métodos públicos)*



Durante el desarrollo de un proyecto software, primero se diseñan los módulos de la solución y a continuación se procede a implementarlos.

IMPORTANT

En el examen es imperativo incluir la descripción del algoritmo.

IMPORTANT

A la hora de diseñar la interfaz de una clase (el conjunto de métodos públicos), intente construir un conjunto minimal de operaciones primitivas que puedan reutilizarse lo máximo posible.

IMPORTANT

Las funciones que realicen un cómputo, no harán también operaciones de E/S



El uso de variables globales puede provocar graves efectos laterales, por lo que su uso está completamente prohibido en esta asignatura.



Salvo casos excepcionales (ver página 565), no definiremos datos miembro públicos. Siempre serán privados.

Esto nos permitirá controlar, desde dentro de la clase, las operaciones que puedan realizarse sobre ellos. Estas operaciones estarán encapsuladas en métodos de la clase.

Desde fuera de la clase, no se tendrá acceso directo a los datos miembro privados. El acceso será a través de los métodos anteriores.



Fomentaremos el uso de datos locales en los métodos de las clases. Sólo incluiremos como datos miembro aquellas características esenciales que determinan una entidad.

Si una propiedad de la clase se puede calcular a partir de los datos miembro, no será a su vez un dato miembro sino que su valor se obtendrá a través de un método de la clase.

La información adicional que se necesite para ejecutar un método, se proporcionará a través de los parámetros de dicho método.



Una consecuencia del Principio de Responsabilidad Única es que jamás mezclaremos en una misma clase métodos de *cómputo* con métodos que accedan a los dispositivos de entrada/salida.

Así pues, tendremos clases que se encargarán únicamente de realizar las E/S de nuestro programa y otras clases para realizar el resto de cálculos.



Principio de Programación:

Ocultación de información (Information Hiding)

Al usar un componente software, no deberíamos tener que preocuparnos de sus detalles de implementación.



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.



Índice alfabético

- álgebra de boole (boolean algebra), 148
- ámbito (scope), 39, 262, 394, 439, 455
- ámbito público (public scope), 475
- ámbito privado (private scope), 475
- índice de variación, 37
- string, 616
- acceso aleatorio (random access), 311
- acceso directo (direct access), 311
- acceso fuera de rango (out of bound access), 318
- algoritmo (algorithm), 3
- ascii, 85
- asociatividad (associativity), 52
- búsqueda binaria (binary search), 331
- búsqueda secuencial (linear/sequential search), 331
- biblioteca (library), 14
- bit, 46
- bucle controlado por condición (condition-controlled loop), 214
- bucle controlado por contador (counter controlled loop), 214
- bucle post-test (post-test loop), 214
- bucle pre-test (pre-test loop), 214
- buffer, 98
- byte, 46
- código binario (binary code), 6
- código fuente (source code), 9
- cabecera (header), 381
- cadena de caracteres (string), 80
- cadena vacía (empty string), 84, 330, 618
- cadena de c (c string), 320
- casting, 64
- clase (class), 452
- code point, 87
- codificación (coding), 87
- cohesión (cohesion), 629
- coma flotante (floating point), 55
- compilación separada (separate compilation), 566
- compilador (compiler), 13
- componentes (elements/components), 309
- componentes léxicos (tokens), 19
- comportamiento (behaviour), 450,

453

comportamiento indeterminado (undefined behaviour), **318, 392, 642**

condición (condition), **121**

conjunto de caracteres (character set), **85**

constante (constant), **33**

constantes a nivel de clase (class constants), **561**

constantes a nivel de objeto (object constants), **561**

constantes estáticas (static constants), **561**

constructor (constructor), **512**

constructor de oficio, **512**

constructor por defecto (default constructor), **520**

cursor (cursor), **98**

dato (data), **3**

datos globales (global data), **439**

datos locales (local data), **394**

datos miembro (data member), **453**

decimal codificado en binario (binary-coded decimal), **57**

declaración (definición) (declaration (definition)), **15**

desbordamiento aritmético (arithmetic overflow), **66**

diagrama de flujo (flowchart), **120**

efectos laterales (side effects), **439**

encapsulación (encapsulation), **453**

entero (integer), **47**

entrada de datos (data input), **16**

enumerado (enumeration), **208**

errores en tiempo de compilación (compilation error), **21**

errores en tiempo de ejecución (execution error), **22**

errores lógicos (logic errors), **22**

espacio de nombres (namespace), **18**

especificador de acceso (access specifier), **455**

estado (state), **453**

estado inválido (invalid state), **531**

estilo camelcase, **29**

estilo snake case, **29**

estilo uppercamelcase, **29**

estructura condicional (conditional structure), **121**

estructura condicional doble (else conditional structure), **137**

estructura repetitiva (iteration/loop), **214**

estructura secuencial (sequential control flow structure), **121**

evaluación en ciclo corto (short-circuit evaluation), **179**

evaluación en ciclo largo (eager evaluation), **179**

excepción (exception), **544**

exponente (exponent), **55**

expresión (expression), **40**

expresiones aritméticas (arithmetic

expression), 64

filtro (filter), 216

flujo de control (control flow), 20, 119

función (function), 16, 43

funciones globales (global functions), 449

funciones miembro (member functions), 453

genéricos (generics), 590

getline, 621

gigo: garbage input, garbage output, 32

google c++ style guide, 30

hardware, 2

identificador (identifier), 15

implementación de un algoritmo (algorithm implementation), 9

indeterminación (undefined), 60

infinito (infinity), 60

instancia (instance), 452

interfaz (interface), 477

iso-10646, 87

iso/iec 8859-1, 86

iteración (iteration), 215

l-value, 41

lógico (boolean), 103

lectura anticipada, 226

lenguaje de programación (programming language), 7

lenguaje ensamblador (assembly language), 7

lenguajes de alto nivel (high level language), 7

leyes de de morgan (de morgan's laws), 148

lista de inicialización del constructor (constructor initialization list), 512

literal (literal), 33

literales de cadenas de caracteres (string literals), 33

literales de caracteres (character literals), 33

literales enteros (integer literals), 48

literales lógicos (boolean literals), 33

literales numéricos (numeric literals), 33

métodos (methods), 450, 453

métodos get, 479

métodos set, 479

módulo (module), 380

macro, 60

mantisa (mantissa), 55

marco de pila (stack frame), 408

memoria dinámica (dynamic memory), 376, 377

modularización (modularization), 448

mutuamente excluyente (mutually exclusive), 145

número de orden (code point), **85**
números mágicos (magic numbers), **35, 116**
números pseudo-aleatorios (pseudo-random numbers), **525**
notación científica (scientific notation), **54**
notación infija (infix notation), **42**
notación prefija (prefix notation), **42**
objeto (object), **452**
objetos (objects), **450**
operador (operator), **16, 43**
operador binario (binary operator), **42**
operador de asignación (assignment operator), **16**
operador de casting (casting operator), **78**
operador de resolución de ámbito (scope resolution operator), **566**
operador n-ario (n-ary operator), **42**
operador unario (unary operator), **42**
ordenación externa, **342**
ordenación interna, **342**
ordenación por inserción (insertion sort), **348**
ordenación por intercambio directo (burbuja) (bubble sort), **351**
ordenación por selección (selection sort), **344**
página de códigos (code page), **85**
parámetros (parameter), **43**
parámetros actuales (actual parameters), **383**
parámetros formales (formal parameters), **383**
pascalcase, **382**
paso de parámetro por valor (pass-by-value), **384**
pila (stack), **408**
plantillas (templates), **590**
precisión (precision), **58**
precondición (precondition), **404**
prevalencia de nombre (name hiding), **263**
principio de programación - ocultación de información (programming principle - information hiding), **420, 653**
principio de programación - sencillez (programming principle - simplicity), **189, 305**
principio de programación - una única vez (programming principle - once and only once), **112, 117**
procedimiento (procedure), **413**
programa (program), **10**
programación (programming), **12**
programación declarativa (declarative programming), **449**
programación estructurada (structured programming), **449**

programación funcional (functional programming), 449

programación modular (modular programming), 449

programación orientada a objetos (object oriented programming), 450

programación procedural (procedural programming), 449

programador (programmer), 2

r-value, 41

rango (range), 45

real (real), 54

redondeo (rounding), 56

registro (record), 556

reglas sintácticas (syntactic rules), 19

salario bruto (gross income), 37

salario neto (net income), 37

salida de datos (data output), 17

sentencia (sentence/statement), 15

sentencia condicional (conditional statement), 122

sentencia condicional doble (else conditional statement), 137

sobrecarga de funciones (function overload), 446

software, 2

struct, 556

tamaño (size), 309

tipo de dato (data type), 15

tipos de datos primitivos (primitive data types), 26

transformación de tipo automática (implicit conversion), 64

uml, 477

unicode, 87

uppercamelcase, 382

usuario (user), 2

valor (value), 16

variables (variables), 33

vector (array), 308

zombi (zombie), 531