



UNIVERSIDAD
DE GRANADA



Departamento de
Ciencias de la Computación
e Inteligencia Artificial

Fundamentos de Programación

Doble Grado en Ingeniería Informática y
Matemáticas

Francisco José Cortijo Bon

cb@decsai.ugr.es

Curso 2021 - 2022



Departamento de Ciencias de la
Computación e Inteligencia Artificial

Universidad de Granada

Fundamentos de Programación

Curso 2021/2022

Apuntes confeccionados por:

- Juan Carlos Cubero Talavera
- Francisco José Cortijo Bon

JC.Cubero@decsai.ugr.es

cb@decsai.ugr.es

**Departamento de Ciencias de la Computación e Inteligencia Artificial
Universidad de Granada.**

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

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

El color rojo se usará para destacar partes especialmente importantes.

Algunos símbolos usados:



Principio de Programación.



denota algo especialmente importante.



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



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



denota código que nos da escalofríos de sólo verlo.



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



denota un consejo de programación.



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



Reseña histórica.



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

Contenidos

I.	Introducción a la Programación	1
I.1.	El ordenador, algoritmos y programas	2
I.1.1.	El Ordenador: Conceptos Básicos	2
I.1.2.	Datos y Algoritmos	3
I.1.3.	Lenguajes de programación	6
I.1.4.	Compilación	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	22
I.2.2.1.	Tokens y reglas sintácticas	22
I.2.2.2.	Palabras reservadas	23
I.2.3.	Tipos de errores en la programación	24
I.2.4.	Cuidando la presentación	26
I.2.4.1.	Escritura de código fuente	26
I.2.4.2.	Etiquetado de las Entradas/Salidas	27

I.3. Datos y tipos de datos	28
I.3.1. Declaración de datos	29
I.3.1.1. Cómo dar un nombre adecuado a los datos	32
I.3.1.2. Inicialización de los datos	34
I.3.1.3. Ámbito de los datos	35
I.3.2. Literales	36
I.3.3. Datos constantes	37
I.4. Operadores y expresiones	42
I.4.1. Expresiones	42
I.4.2. Terminología en Matemáticas	44
I.4.3. Operadores en Programación	45
I.5. Tipos de datos simples en C++	47
I.5.1. Los tipos de datos enteros	48
I.5.1.1. Representación de los enteros	48
I.5.1.2. Rango de los enteros	49
I.5.1.3. Literales enteros	50
I.5.1.4. Operadores	51
I.5.1.5. Expresiones enteras	53
I.5.1.6. Desbordamiento	55
I.5.2. Los tipos de datos reales	58
I.5.2.1. Literales reales	58

I.5.2.2.	Representación de los reales	59
I.5.2.3.	Rango y Precisión	61
I.5.2.4.	Operadores	64
I.5.2.5.	Funciones estándar	65
I.5.2.6.	Expresiones reales	67
I.5.3.	Operando con tipos numéricos distintos	69
I.5.3.1.	Asignaciones a datos de expresiones de distinto tipo	69
I.5.3.2.	Expresiones con datos numéricos de distinto tipo	74
I.5.3.3.	El operador de casting (Ampliación)	79
I.5.4.	El tipo de dato carácter	81
I.5.4.1.	Representación de caracteres en el ordenador	81
I.5.4.2.	Tratamiento de caracteres en C++	84
I.5.4.3.	Literales de carácter	89
I.5.4.4.	Funciones estándar y operadores	91
I.5.5.	El tipo de dato cadena de caracteres	92
I.5.5.1.	Literales de cadena de caracteres	93
I.5.5.2.	Operaciones con datos string	94
I.5.5.3.	Operaciones de conversión a <code>string</code> / desde <code>string</code>	97
I.5.5.4.	Lectura de datos <code>string</code> : la función <code>getline</code>	98

I.5.6. El tipo de dato lógico o booleano	100
I.5.6.1. Rango. Representación	100
I.5.6.2. Funciones estándar y operadores lógicos .	102
I.5.6.3. Operadores Relacionales	104
I.5.7. Lectura de varios datos	107
I.6. El principio de una única vez	112
II. Estructuras de control	117
II.1. Estructura condicional	118
II.1.1. Flujo de control	118
II.1.2. Estructura condicional simple	121
II.1.2.1. Formato	121
II.1.2.2. Diagrama de flujo	123
II.1.2.3. Cuestión de estilo	129
II.1.2.4. Condiciones compuestas	130
II.1.2.5. Variables no asignadas en los condicionales	132
II.1.2.6. Estructuras condicionales consecutivas .	134
II.1.3. Estructura condicional doble	137
II.1.3.1. Formato	137
II.1.3.2. Condiciones mutuamente excluyentes .	144
II.1.3.3. Estructuras condicionales dobles consecutivas	149

II.1.4. Anidamiento de estructuras condicionales	155
II.1.4.1. Funcionamiento del anidamiento	155
II.1.4.2. Anidar o no anidar: he ahí el dilema	162
II.1.5. Estructura condicional múltiple	170
II.1.6. Algunas cuestiones sobre condicionales	173
II.1.6.1. Álgebra de Boole	173
II.1.6.2. Cuidado con la comparación entre reales . .	177
II.1.6.3. Evaluación en ciclo corto y en ciclo largo . .	179
II.1.7. Programando como profesionales	180
II.1.7.1. Diseño de algoritmos fácilmente extensibles	180
II.1.7.2. Descripción de un algoritmo	190
II.1.7.3. Descomposición de una solución en tareas más sencillas	194
II.1.7.4. Las expresiones lógicas y el principio de una única vez	197
II.1.7.5. Separación de entradas/salidas y cómputos	199
II.1.7.6. El tipo enumerado y los condicionales . . .	205
II.2. Estructuras repetitivas	212
II.2.1. Bucles controlados por condición: pre-test y post-test	212
II.2.1.1. Formato	212
II.2.1.2. Algunos usos de los bucles	214
II.2.1.3. Bucles con lectura de datos	222

II.2.1.4. Bucles sin fin	234
II.2.1.5. Condiciones compuestas	236
II.2.1.6. Bucles que buscan	240
II.2.2. Programando como profesionales	243
II.2.2.1. Evaluación de expresiones dentro y fuera del bucle	243
II.2.2.2. Bucles que no terminan todas sus tareas . .	245
II.2.2.3. Estilo de codificación	249
II.2.3. Bucles controlador por contador	250
II.2.3.1. Motivación	250
II.2.3.2. Formato	252
II.2.4. Anidamiento de bucles	259
II.3. Particularidades de C++	269
II.3.1. Expresiones y sentencias son similares	269
II.3.1.1. El tipo <code>bool</code> como un tipo entero	269
II.3.1.2. El operador de asignación en expresiones .	271
II.3.1.3. El operador de igualdad en sentencias . .	272
II.3.1.4. El operador de incremento en expresiones .	273
II.3.2. El bucle <code>for</code> en C++	275
II.3.2.1. Bucles <code>for</code> con cuerpo vacío	275
II.3.2.2. Bucles <code>for</code> con sentencias de incremento incorrectas	275

II.3.2.3. Modificación del contador	276
II.3.2.4. El bucle <code>for</code> como ciclo controlado por condición	279
II.3.3. Otras (perniciosas) estructuras de control	287
III. Funciones	289
III.1. Fundamentos	290
III.1.1. Las funciones realizan una tarea	290
III.1.2. Definición	292
III.1.3. Parámetros formales y reales	293
III.1.4. Ámbito de un dato. Datos locales	305
III.1.5. La Pila	315
III.2. El principio de ocultación de información	320
III.3. Funciones void	324
III.4. Ámbito de un dato (revisión)	328
III.5. Parametrización de funciones	334
III.6. Programando como profesionales	341
III.6.1. Cuestión de estilo	341
III.6.2. Diseño de la cabecera de una función	344
III.6.3. Precondiciones	354
III.6.4. Documentación de una función	355
IV. Registros, vectores y matrices	359

IV.1. Registros	360
IV.1.1. El tipo de dato struct	360
IV.1.2. struct y funciones	362
IV.1.2.1. NOTA: Funciones que devuelven struct	363
IV.1.3. Ámbito de un struct	367
IV.1.4. Inicialización de los campos de un struct	368
IV.2. Vectores	369
IV.2.1. Introducción	369
IV.2.1.1. Motivación	369
IV.2.1.2. Declaración	371
IV.2.2. Operaciones básicas	372
IV.2.2.1. Acceso	372
IV.2.2.2. Asignación	374
IV.2.2.3. Lectura y escritura	375
IV.2.2.4. Inicialización	376
IV.2.3. Representación en memoria	377
IV.2.4. Gestión de componentes utilizadas	380
IV.3. Recorridos sobre vectores	390
IV.3.1. Algoritmos de búsqueda	390
IV.3.1.1. Búsqueda Secuencial	391
IV.3.1.2. Búsqueda Binaria	393

IV.3.1.3. Otras búsquedas	396
IV.3.2. Recorridos que modifican componentes	399
IV.3.2.1. Inserción de un valor	399
IV.3.2.2. Eliminación de un valor	400
IV.3.3. Algoritmos de ordenación	402
IV.3.3.1. Ordenación por Selección	404
IV.3.3.2. Ordenación por Inserción	408
IV.3.3.3. Ordenación por Intercambio Directo (Método de la Burbuja)	411
IV.4. Matrices	416
IV.4.1. Declaración y operaciones con matrices	416
IV.4.1.1. Declaración	416
IV.4.1.2. Acceso y asignación	417
IV.4.1.3. Inicialización	418
IV.4.2. Representación en memoria (Ampliación)	419
IV.4.3. Matrices con más de dos dimensiones	420
IV.4.4. Gestión de componentes útiles con matrices	421
IV.4.4.1. Se usan todas las componentes	421
IV.4.4.2. Se ocupan todas las columnas pero no todas las filas (o al revés)	423
IV.4.4.3. Se ocupa un bloque rectangular	425

IV.4.4.4. Se ocupan las primeras filas, pero con tamaños distintos	435
V. Clases	437
V.1. Motivación. Clases y Objetos	438
V.2. Encapsulación	443
V.2.1. Datos miembro	445
V.2.2. Métodos	449
V.2.3. Controlando la integridad de los objetos	457
V.2.4. Llamadas entre métodos dentro del propio objeto . .	459
V.3. Ocultación de información	463
V.3.1. Ámbito público y privado	463
V.3.2. Datos miembro privados	466
V.3.3. Métodos privados	477
V.4. Constructores	482
V.4.1. Estado inválido de un objeto	482
V.4.2. Definición de constructores	485
V.4.3. Constructores sin parámetros	492
V.4.4. Sobrecarga de constructores	497
V.4.5. Llamadas entre constructores	499
V.4.6. Estado inválido de un objeto -revisión-	500
V.5. Copiando objetos	503

V.5.1. Operador de asignación	503
V.5.2. El constructor de copia	505
V.6. Datos miembro constantes	508
V.6.1. Constantes a nivel de objeto	510
V.6.2. Constantes a nivel de clase	513
V.6.3. El operador de asignación y el constructor de copia en clases con datos miembro constantes	515
V.7. Programando como profesionales	517
V.7.1. Datos miembro vs parámetros de los métodos	518
V.7.2. Principio de Responsabilidad única y cohesión de una clase	527
V.7.3. Funciones vs Clases	534
V.7.4. Comprobación y tratamiento de las precondiciones .	537
V.8. Vectores y clases	549
V.8.1. Los vectores como datos miembro	549
V.8.2. Comprobación de las precondiciones	560
V.8.3. Vectores como datos locales de un método	563
V.8.4. Vectores como parámetros a un método	565
V.8.5. Vectores de objetos	566
V.9. La clase Secuencia de caracteres	568
V.9.1. Métodos básicos	568
V.9.2. Métodos de búsqueda	577

V.9.3. Llamadas entre métodos	594
V.9.4. Algoritmos de ordenación	595
V.10. La clase string	602
V.10.1. Métodos básicos	602
V.10.2. Lectura de un string con getline	605
V.10.3. El método ToString	606
VI. Clases (Segunda parte)	607
VI.1. Métodos y objetos	608
VI.1.1. Objetos y funciones	608
VI.1.2. Objetos locales de un método	610
VI.1.3. Pasando objetos como parámetros a los métodos	613
VI.1.4. Métodos que devuelven objetos	620
VI.1.5. Operaciones binarias entre objetos de una misma clase	632
VI.1.6. Acceso a los datos miembros privados	644
VI.2. Objetos como datos miembro de otros objetos	648
VI.3. Tablas de datos	660
VI.3.1. Representación de una tabla con matrices	661
VI.3.1.1. Tabla rectangular usando una matriz	661
VI.3.1.2. Tabla dentada usando una matriz	672
VI.3.2. Representación de una tabla con vectores de objetos	682
VI.3.2.1. Tabla rectangular usando un vector de objetos	682

VI.3.2.2. Tabla dentada usando un vector de objetos	693
VI.4. Diseño de una solución (Ampliación)	701
VI.4.1. Funciones globales versus métodos	701
VI.4.2. Fábricas de objetos (Ampliación)	706
VI.5. Tratamiento de errores con excepciones	710
VI.6. Ciclo de vida del software (Ampliación)	722
 VII. Recursividad	 729
VII.1. El concepto matemático de Recursividad	730
VII.1.1. Soluciones recursivas	730
VII.1.2. Diseño de algoritmos recursivos	733
VII.2. Funciones recursivas	737
VII.2.1. Definición de funciones recursivas	737
VII.2.2. Ejecución de funciones recursivas	738
VII.3. Clases con métodos recursivos	747
VII.3.1. Métodos recursivos	747
VII.3.2. Ordenación con Quicksort (Ampliación)	759
VII.4. Recursividad versus iteración	765

Tema I

Introducción a la Programación

Objetivos:

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

Autor: Juan Carlos Cubero.

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

I.1. El ordenador, algoritmos y programas

I.1.1. El Ordenador: Conceptos Básicos

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



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

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

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

Ejemplo. Algoritmo de la media aritmética de N valores.

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

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

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

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

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

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

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

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

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

- ▷ **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.

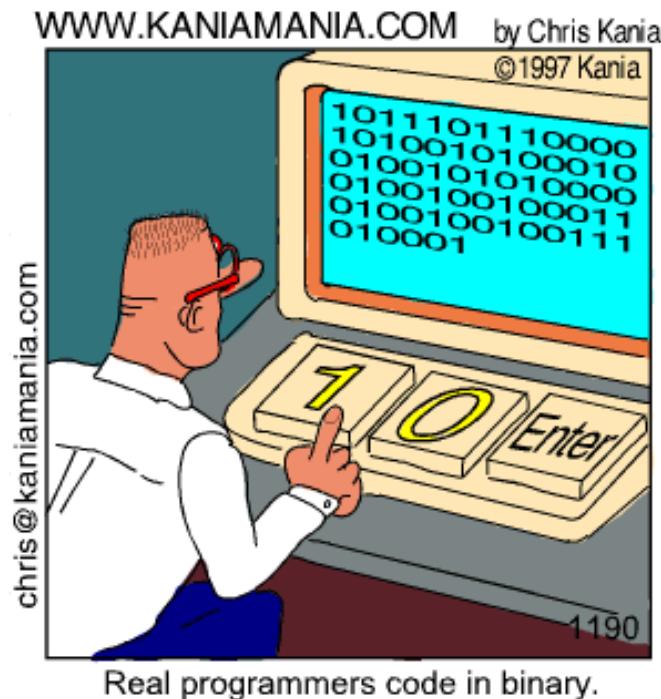
Una vez diseñado el algoritmo, debemos escribir las órdenes que lo constituyen en un lenguaje que entienda el ordenador.

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



I.1.3. Lenguajes de programación

Código binario (Binary code) :



"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 Standardization Organization <http://www.iso.org>

Comité técnico JTC1: Information Technology

Subcomité SC-22: Programming languages, their environments and system software interfaces.

Working Group 21: C++

<http://www.open-std.org/jtc1/sc22/wg21/>

2011 Revisión del estándar con importantes cambios.

2014 Última revisión del estándar con cambios menores.

2017? Actualmente en desarrollo la siguiente versión C++ 17.

¿Qué programas se han hecho en C++?

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

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

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 calcular el mayor *valor* guardado en un vector v entre las casillas pos_inicial y pos_final:

```
// Suponemos que:  
// 1) v existe y tiene valores almacenados.  
// 2) pos_inicial y pos_final contienen valores correctos.  
  
menor = v[pos_inicial];  
pos_menor = pos_inicial;  
  
for (pos = pos_inicial+1; pos <= pos_final; pos++){  
    if (v[pos] < menor) {  
        menor = v[pos];  
        pos_menor = pos;  
    }  
}
```

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. Calcular la hipotenusa de un triángulo rectángulo.

Pitagoras.cpp

```
/****************************************/
// FUNDAMENTOS DE PROGRAMACIÓN
//
// (C) FRANCISCO JOSÉ CORTIJO BON
// DPTO. DE CIENCIAS DE LA COMPUTACIÓN E INTELIGENCIA ARTIFICIAL
//
/* Programa para calcular la hipotenusa de un triángulo.
   Implementa el algoritmo de Pitágoras
```

Entradas: los catetos de un triángulo rectángulo (lado1, lado2).

Salidas: La hipotenusa (hip).

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

*/

```
/****************************************/
```

```
#include <iostream>    // Inclusión de los recursos de E/S
#include <cmath>        // Inclusión de los recursos matemáticos

using namespace std;
```

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

    // Entrada de datos

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

    // Cálculos

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

    // Salida (presentación de resultados)

    cout << endl;
    cout << "La hipotenusa vale " << hip << endl ;

    return (0);
}
```

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.

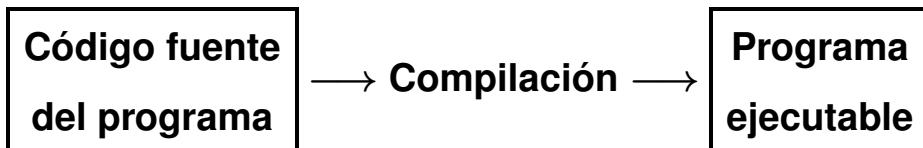
Ejemplo. Programa para dibujar planos de pisos.

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;
    .....
}
```

→ Compilador →

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.

Los programas en C++ pueden dividirse en varios ficheros aunque en esta asignatura escribiremos programas que contienen todo el código en un único fichero. En otras asignaturas (*Metodología de la Programación*) aprenderá a construir proyectos en los que el código se distribuye en diferentes ficheros.

Usaremos el programa del fichero Pitagoras.cpp mostrado en la sección I.1.3 para explicar con cierto detalle la estructura de un sencillo programa en C++.

I.2.1. Organización de un programa

- ▷ 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. Es útil para los programadores para informar a otros acerca de qué hace el código y cómo lo hace.

- ▷ Al principio del fichero, tras los comentarios de presentación, se indica que vamos a usar una serie de recursos *definidos* en un fichero externo o *biblioteca (library)* y *declarados* en un *fichero de cabecera (header)*.

Los ficheros de cabecera son ficheros de texto y tienen extensión .h mientras que los ficheros de biblioteca son ficheros binarios

(realmente son paquetes de módulos objeto -de extensión .o-) y tienen extensión .a aunque puede encontrarlos también con otras extensiones (.lib, .dll) pero no entraremos en detalles.

Las líneas

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

permiten “incluir” el contenido de los ficheros `iostream.h` y `cmath.h` en el punto en el que están escritas: se sustituye cada línea `#include` por el contenido del fichero indicado.

La primera línea `include` permite que pueda usarse el teclado (`cin`) para la entrada de datos y el terminal (`cout`) para la presentación de mensajes y valores de datos. En sistemas windows `cout` se asocia al llamado *Símbolo del Sistema, Command Prompt* ó *CMD*. El *CMD* se puede utilizar para escribir órdenes y ejecutarlas, lo que puede ser especialmente útil para automatizar tareas mediante secuencias de órdenes en archivos batch (`.bat`) para llevar a cabo funciones administrativas avanzadas.

Las declaraciones de `cin` y `cout` se encuentran en el fichero `iostream.h`. Escribir la línea `#include <iostream>` permite que el compilador pueda procesar las declaraciones contenidas en `iostream.h` y que nuestro programa pueda usar los elementos declarados en él. Si la instalación del compilador está bien hecha no deberíamos preocuparnos del lugar en el que se encuentra el fichero `iostream.h`.

La segunda línea `include` permite que puedan usarse funciones matemáticas avanzadas como `sqrt` (square root ó raíz cuadrada), `pow` (power ó potencia), `sin` (sinus ó seno), ...

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 `}`.

La función `main` es el punto de entrada para la ejecución del programa. Esto significa que cuando se ejecuta un programa la primera instrucción en ser ejecutada es la primera instrucción de la función `main`¹.

- ▷ Dentro del programa principal se escriben órdenes o instrucciones. Una *orden (sentence/statement)* es una parte del código fuente que el compilador traduce en una instrucción en código binario. Éstas 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.

La manera correcta -formalmente- de finalizar la ejecución de un programa es escribir la orden `return 0`. De esta manera indicamos que el programa termina devolviendo el valor `0` (todo ha sido correcto) al proceso que inicia la ejecución del programa. No obstante puede obviarse esta instrucción y el compilador la asumirá por defecto.

¹excepto si hubieran declaraciones de datos globales (que serán las primeras instrucciones en ser ejecutadas)

Veamos algunos tipos de órdenes:

▷ **Órdenes de declaración de datos**

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

Debemos asociar cada dato a un único **tipo de datos (data type)**. El tipo de dato determina los valores que podemos asociarle y las operaciones que podemos realizar.

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

Cada dato que se desee usar en un programa debe declararse previamente. Por ahora, lo haremos al principio (después de `main`),

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

Declara tres datos de tipo real (doble precisión) que el programador puede usar en el programa. Su valor es indefinido (arbitrario, aleatorio) después de la declaración.

También podrían haber sido declaradas en una sola línea:

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

▷ **Órdenes 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 cuantas veces queramos.

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

▷ **Órdenes de entrada de datos**

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

Las órdenes 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 declarado en `iostream.h` sobre el que se puede emplear el operador binario `>>`

Por ejemplo, al ejecutarse la orden

```
cin >> lado1;
```

el programa espera a que el usuario introduzca un valor real (`double`) 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 escri-

biendo el valor, éste se muestra en pantalla (el eco), incluyendo el salto de línea.

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

▷ **Órdenes de salida de datos**

Por otra parte, las órdenes 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 declarado en `iostream.h` sobre el que se puede emplear el operador binario `<<`

```
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.;"
```

El resultado sería:

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

Preferimos no emplear el carácter especial `\n` para forzar un salto de línea y hacerlo enviando a `cout` el manipulador `endl`:

```
cout << "Bienvenido. Salto a la siguiente linea." << endl;  
cout << endl << "Empiezo en una nueva linea.;"
```

El resultado -visualmente- sería el mismo².

²En el caso de `endl` se fuerza a que se vacíe el *buffer* de salida, que es interesante para los programadores noveles cuando están depurando sus programas

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

```
cout << 3.1415927;
```

- Si escribimos el *nombre* de un dato, se imprime su *contenido* de forma textual (con caracteres).

```
cout << hipotenusa;
```

Podemos formar secuencias de texto complejas concatenando diferentes elementos en una única sentencia usando el operador `<<` para diferenciarlos:

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

Realmente, habría que anteponer el **espacio de nombres (namespace)** std antes de usar los identificadores cout y cin:

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

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

Los namespaces sirven para organizar los recursos (funciones, clases, etc.) ofrecidos por el compilador o construidos 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 espacio de nombres estándar (std)

Con estas primeras indicaciones, que serán ampliadas y matizadas a lo largo del curso, la estructura básica de un programa es la siguiente³:

```
[ /* Comentarios de cabecera
    Breve descripción en lenguaje natural
    de lo que hace el programa */ ]
[ líneas include]
[ using namespace std; ]
int main(){
    [Declaración de datos]
    [Órdenes del programa separadas por ;]
    [return 0; ]
}
```

³Los corchetes delimitan secciones opcionales

I.2.2. Elementos básicos de un lenguaje de programación

I.2.2.1. Tokens y reglas sintácticas

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

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

main ; (== = hipotenusa * /*

Pero por ejemplo, ni 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++

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

Palabras reservadas adicionales de C++

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

I.2.3. Tipos de errores en la programación

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



▷ **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 <iostre am>

USING namespace std;

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

    2 = lado1;
    lado1 = 2
    hipotenusa = sqrt(lado1*lado1 + ladp2*ladp2);
    cout << "La hipotenusa vale << hipotenusa;
()
```

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

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

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



▷ ***Errores lógicos (Logic errors)***

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

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

I.2.4. Cuidando la presentación

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

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

I.2.4.1. Escritura de código fuente

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

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

Este código fuente genera el mismo programa que el de la página 10 pero es mucho más difícil de leer.

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



I.2.4.2. Etiquetado de las Entradas/Salidas

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

```
int totalVentas = 45;   
int numeroVentas = 78;  
cout << totalVentas << numeroVentas; // Imprime 4578  
  
cout << endl;  
  
cout << "Suma total de ventas = " << totalVentas << endl;   
cout << "Número total de ventas = " << numeroVentas << endl;  
cout << endl;
```

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

Tanto las instrucciones como los datos son combinaciones adecuadas de 0 y 1. Los datos y las instrucciones de un programa en ejecución están en memoria. En localizaciones diferentes, pero siempre en memoria.

Datos

"Juan Pérez" →

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

75225813 →

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

3.14159 →

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

Instrucciones

Abrir Fichero →

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

Imprimir →

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

Nos centramos en los datos.

I.3.1. Declaración de datos

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

Nombre de empleado: Juan Pérez

Número de habitantes : 75225813

π : 3.14159

Un programa necesitará representar información de diverso tipo (cadenas de caracteres, enteros, reales, etc) Cada lenguaje de programación ofrece sus propios tipos de datos, denominados **tipos de datos primitivos (primitive data types)**. Por ejemplo, en C++: string, int, double, etc. Además, el programador podrá crear sus propios tipos usando otros recursos, como por ejemplo, las *clases* (ver tema III).

Declaración de un dato (data declaration) : Es la orden en la que se asocia un *nombre* y un *tipo de dato* a un dato. El valor que se le puede asignar a un dato y las operaciones que pueden realizarse con él dependen de su tipo.

El tipo de dato restringe los valores que puede tomar un dato y las operaciones que pueden realizarse con él.

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

```
string nombre_empleado;
int     numero_habitantes;
double pi;

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

nombre_empleado	numero_habitantes	pi
Juan Pérez	75225	3.1415927

Como indicamos anteriormente podemos declarar varias variables de un mismo tipo en la misma línea. Basta separarlas con coma:

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

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

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

```
double dato = 4.5;
```

equivale a:

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

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

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

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

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

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

I.3.1.1. Cómo dar un nombre adecuado a los datos

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, 11, 12, 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 estilos como *camelCase*: precioVentaPublico, tasaAnual o *UpperCamelCase*: PrecioVentaPublico, TasaAnual **Este último es el que usaremos para las funciones y métodos.**

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



pvp, tae, iva

- ▷ No se nombrarán dos datos con identificadores que difieran únicamente en la capitalización, o un sólo carácter.



cuenta, cuentas, Cuenta



cuenta, colección_cuentas, cuenta_ppal

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

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



Clase CuentaBancaria, **Método** Ingresar

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

IMPORTANT

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

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

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

I.3.1.2. 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.

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

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

    salario_bruto      retencion
    [?]                [?]

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

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

salario_bruto	retencion
32538.0	4229.94

Un error **Lógico** muy común es usar un dato no asignado:

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



Imprimirá un valor indeterminado.

GIGO: Acrónimo de “Garbage In, Garbage Out”

Indica que entradas incorrectas (datos inválidos, fuera de rango, con formato desconocido,... provocan, irremediablemente, resultados incorrectos (no solamente errores de ejecución sino valores incorrectos).

I.3.1.3. Ámbito de los datos

El **ámbito (scope)** de un dato *v* es el conjunto de todos aquellos elementos de código (bloques, funciones, módulos...) 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.3.2. Literales

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

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

I.3.3. Datos constantes

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

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

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

Ejemplo. 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/~carlos/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:

- ▷ El nombre dado a la constante (`PI`) proporciona más información al programador y hace que el código sea más legible.
Esta ventaja podría haberse conseguido usando un dato variable, pero entonces podría cambiarse su valor por error dentro del código. Al ser constante, su modificación no es posible.
- ▷ Código menos propenso a errores. Para cambiar el valor de `PI` (a 3.1415927, por ejemplo), sólo hay que modificar la línea de la declaración de la constante.
Si hubiésemos usado literales, tendríamos que haber recurrido a un cut-paste, muy propenso a errores.

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

Ejercicio. Modifique el ejemplo de la página 34 introduciendo una constante que contenga el valor del porcentaje de la retención (0.18).

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

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

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

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

    retencion = salario_bruto * PORCENTAJE_RETENCION;

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

salario_bruto	PORCENTAJE_RETENCION	retencion
32538.0	0.18	4229.94

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

Una sintaxis de programa algo más completa:

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

I.4. Operadores y expresiones

I.4.1. Expresiones

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

3

3 + 5

lado1

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

lado1 * lado1

lado2 * lado2

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

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

Una expresión NO es una sentencia de un programa:

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

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

3 + 5 = lado1; // Error de compilación

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

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

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

3 + 5 es una expresión entera

3.5 + 6.7 es una expresión de reales

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

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

Imprime en pantalla:

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

A lo largo del curso justificaremos que es mejor no incluir expresiones dentro de las instrucciones cout (más detalles en la página 115). 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. Ejemplos:

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

Los operadores binarios suelen ser infixos:

3 + 5

lado * lado

- ▷ Los definidos en bibliotecas externas.

Por ejemplo, `cmath`

Suelen usarse tokens alfanuméricos para su representación. Ejemplos:

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

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

`sqrt(4.2)`

`sin(6.4)`

`pow(3 , 6)`

Tradicionalmente se usa el término **operador (operator)** a secas para de-

notar los primeros, y el término **función (function)** para los segundos.

A los argumentos de las funciones se les denomina **parámetros (parameter)**.

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

```
double dato;  
  
dato = 4;           // dato contiene 4  
dato = dato + 3;   // dato contiene 7
```

En una sentencia de asignación

`variable = <expresión>`

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

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

$$F = m * a$$

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

El comportamiento de un tipo de dato viene dado por:

- ▷ El *rango (range)* de valores que puede representar, que depende de la cantidad de memoria que dedique el compilador a su representación interna. Intuitivamente, 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.

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. Esta representación de un entero es válida en cualquier lenguaje de programación. Para el ejemplo anterior:

0000000000110101

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**

$0000000000110101 \neq 0000000000110110$

- ▷ **Número de datos distintos representables = 2^r**

I.5.1.2. Rango de los enteros

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

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

El estándar de C++ no obliga a los compiladores a usar un tamaño determinado.

Lo usual es que un int ocupe 32 bits. El rango sería:

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

```
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:

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

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

I.5.1.3. Literales enteros

Como ya vimos en la página 36, 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

+ - * / %

suma, resta, producto, división entera y módulo.

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

Binarios. Notación infija: a*b

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

Operaciones usuales:

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

Operadores unarios de incremento y decremento

++ y -- Unarios de notación postfija.

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

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

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

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

Operador unario de cambio de signo

- **Unario de notación prefija.**

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

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

I.5.1.5. Expresiones enteras

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

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

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

Reglas de precedencia:

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

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

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

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

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

Ante la duda, 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. 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.

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

I.5.1.6. Desbordamiento

¿Qué ocurre en el siguiente código?

```
int main(){
    int entero;

    entero = 2147483647 + 1;    // máximo int representable + 1
                                // Resultado: -2147483648
    cout << entero;
}
```

En Matemáticas, $2147483647 + 1$ es 2147483648 , pero en el código de C++, los literales 2147483647 y 1 son ambos `int`. Por lo tanto, el compilador entiende que la expresión $2147483647 + 1$ es también `int`, pero el resultado se sale del rango representable en un `int`.

Se produce un error lógico denominado **desbordamiento aritmético (arithmetic overflow)** y el resultado de evaluar la expresión $2147483647 + 1$ es un valor indeterminado. En este caso, el valor concreto es -2147483648 .

Hay que destacar que no es un error de ejecución sino lógico. Son errores difíciles de detectar con los que habrá que tener especial cuidado.

Cuando una expresión numérica da como resultado un valor que no puede representarse, se produce un desbordamiento aritmético y el resultado es un número indeterminado.

¿Qué ocurre en el siguiente código?

```
int main(){
    int entero;
    long grande;

    entero = 2147483647;    // máximo int representable
    grande = entero + 1;    // -2147483648 Desbordamiento

    cout << entero_grande;
}
```

Sigue habiendo un desbordamiento ya que éste se produce al evaluar la expresión $2147483647 + 1$ que resulta en -2147483648 . A continuación se le asigna este negativo a un `long` (que obviamente está en su rango)

Posteriormente se analizará este caso con más detalle (ver página 69).

Ampliación:

Para obtener directamente los límites de los rangos de cada tipo, puede usarse la biblioteca 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.5.2. Los tipos de datos reales

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

- ▷ **Parte entera de** 4,56 → 4
- ▷ **Parte real de** 4,56 → 56

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

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

I.5.2.1. Literales reales

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

800.457 4.0 -3444.5

Importante:

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

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

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

5.32e+5 representa el número $5,32 \times 10^5 = 532000$

42.9e-2 representa el número $42,9 \times 10^{-2} = 0,429$

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:

$$\begin{aligned} 42,001 &\rightarrow \text{valor} = 4,2001 \quad \text{escala} = 10 \\ 42001 &\rightarrow \text{valor} = 4,2001 \quad \text{escala} = 10^4 \\ 0,42001 &\rightarrow \text{valor} = 4,2001 \quad \text{escala} = 10^{-1} \end{aligned}$$

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

En la representación en coma flotante, la escala es 2. A *grossso 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 los reales pueden devolver valores que sólo sean aproximados!



Especial cuidado tendremos con operaciones del tipo

Repite varias veces

Ir sumándole a una variable_real varios valores reales;



ya que los errores de aproximación se irán acumulando.

I.5.2.3. Rango y Precisión

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

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

C++ ofrece varios tipos reales: float y double (y long double a partir de C++11). Con 64 bits, pueden representarse exponentes hasta ± 308 .

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

Tipo	Tamaño	Rango	Precisión
float	4 bytes	$+\/-3.4 \times 10^{-38}$	7 dígitos aproximadamente
double	8 bytes	$+\/-1.7 \times 10^{-308}$	15 dígitos aproximadamente

En resumen:

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

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

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.5.2.4. Operadores

Los operadores matemáticos usuales también se aplican sobre datos reales:

+, -, *, /

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

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

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

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

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

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

I.5.2.5. Funciones estándar

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

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

Todos los anteriores son unarios excepto `pow`, que es binario (base, exponente). Devuelven un real.

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

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

using namespace std;

int main(){
    double real, otro_real;

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

Nota:

Observe que una misma función (`abs` p.e.) 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.

C++ proporciona una serie de funciones (también hay que incluir `cmath`) para conseguir un número real sin decimales a partir de otro:

- ▷ `round()`: redondea al valor más cercano.
- ▷ `trunc()`: descarta los decimales, sin otra consideración.
- ▷ `floor()` y `ceil()`: devuelven el valor sin decimales inferior y superior, respectivamente.

En la tabla siguiente puede apreciar el efecto de estas funciones sobre diferentes valores `double` positivos y negativos:

valor	round	floor	ceil	trunc
2.3	2.0	2.0	3.0	2.0
3.8	4.0	3.0	4.0	3.0
5.5	6.0	5.0	6.0	5.0
-2.3	-2.0	-3.0	-2.0	-2.0
-3.8	-4.0	-4.0	-3.0	-3.0
-5.5	-6.0	-6.0	-5.0	-5.0

I.5.2.6. Expresiones reales

Son expresiones cuyo resultado es un número real.

`sqrt(real)` es una expresión real

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

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

Precedencia de operadores en las expresiones reales:

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

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



Ejercicio. Construya una expresión para calcular la siguiente fórmula:

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

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

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

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

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

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

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

$$x_o + vt$$

dónde x_0 es la posición inicial, v la velocidad y t el tiempo transcurrido. Suponga que los tres datos son reales.

I.5.3. Operando con tipos numéricos distintos

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

Vamos a analizar la siguiente situación:

```
variable_del_tipo_A = expresión_con_datos_del_tipo_B;
```

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

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

En segundo lugar, se realiza la asignación. Si el tipo del resultado obtenido en la parte derecha de la asignación es distinto al del dato de la parte izquierda, el compilador realiza una *transformación de tipo (casting)* de la expresión de la derecha al tipo de dato de la parte izquierda de la asignación.

Esta transformación es temporal (mientras se evalúa la expresión)

Si el resultado de la expresión no cabe en la parte izquierda, se produce un desbordamiento aritmético. El valor asignado será un valor indeterminado.

Veamos varios casos:

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

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

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

grande = chico;     // chico int -> grande long long
                     // long long = long long
cout << chico;      // chico sigue siendo int
```



La transformación de int a long long es inmediata:

Otro ejemplo:

```
double real;  
int entero;  
  
entero = 5;  
real    = entero;    // 5 int -> 5 double  
                  // double = double  
cout << entero;    // entero sigue siendo int  
                  // Imprime 5 y no 5.0
```

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

```
int chico;
```

```
long long grande;
```

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

```
grande = 6000000; // 6000000 es un literal int
```

```
// 6000000 int -> 6000000 long long
```

```
// long long = long long
```

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

```
// El resultado 6000000 cabe en chico
```



```
// int = int
```

```
grande = 360000000000000;
```

```
// 360000000000000 es un literal long long
```

```
// long long = long long
```



```
chico = grande; // 360000000000000 no cabe en un int
```

```
// Desbordamiento.
```

```
// chico = -415875072
```

```
// int = int
```

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

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

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

    return 0;
}
```

I.5.3.2. 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 de la expresión.

Esta transformación es temporal (mientras se evalúa la expresión)

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
```



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

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

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

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



La expresión `edad1 + edad2` es entera y devuelve 15. Por tanto, los dos operandos de la expresión `(edad1 + edad2)/2` son enteros, por lo que el operador de división actúa sobre enteros y es la división entera, devolviendo el entero 7. Al asignarlo a la variable real `media`, se transforma en 7.0. Se ha producido un error lógico.

Possibles soluciones (de peor a mejor)

▷ Usar un dato temporal de un tipo mayor. 

```
int edad1 = 10, edad2 = 5;
double media, edad1_tmp;

edad1_tmp = edad1;
media      = (edad1_tmp + edad2)/2;
                // double + int es double
                // double / int es double
                // media = 7.5
```

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

▷ Cambiar el tipo de dato original de las variables. 

```
double edad1 = 10, edad2 = 5;  
double media;  
  
media = (edad1 + edad2)/2;  
// double + double es double  
// double / int es double  
// media = 7.5
```

Debemos evitar esta solución ya que el tipo de dato asociado a las variables debe depender de la semántica de éstas. En cualquier caso, cabe la posibilidad de reconsiderar la decisión con respecto a los tipos asociados y cambiarlos si la semántica de las variables así lo demanda. No es el caso de nuestras variables de edad, que son enteras.

▷ Usar un casting manual tal y como se indica en la sección I.5.3.3

(página 79) 

▷ Forzamos la división real introduciendo un literal real:

```
int edad1 = 10, edad2 = 5;  
double media;  
  
media = (edad1 + edad2)/2.0;   
// int + int es int  
// int / double es double  
// media = 7.5
```

Ejemplo. ¿Qué pasaría en este código?

```
int chico = 1234567890;  
long long grande;  
  
grande = chico * chico;  
  
// grande = 304084036      Error lógico
```



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

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 (recordad lo visto en la página 46)

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

Nota:

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

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

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

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

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

I.5.3.3. El operador de casting (Ampliación)

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

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

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

Ejemplo. Media aritmética:

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



Ejemplo. Retomamos el ejemplo de la página 77:

```
int chico = 1234567890;  
long long grande;  
  
grande = static_cast<long long>(chico) * chico; // chico int -> chico long long  
// long long * int  
// grande = 1524157875019052100  
  
// chico sigue siendo int después de la instrucción anterior
```



¿Por qué no es correcto lo siguiente?

```
int chico = 1234567890;  
long long grande;  
  
grande = static_cast<long long> (chico * chico);  
// grande = 304084036
```



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

(<tipo de dato>) expresión

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

I.5.4. El tipo de dato carácter

I.5.4.1. Representación de caracteres en el ordenador

Frecuentemente, querremos manejar información que podemos representar con un único carácter. Por ejemplo, grupo de teoría de una asignatura, carácter a leer desde el teclado para seleccionar una opción de un menú, calificación obtenida (según la escala ECTS), tipo de moneda, etc. Pueden ser tan básicos como las letras del alfabeto inglés, algo más particulares como las letras del alfabeto español o complejos como los jeroglíficos 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	!	A	97	a	129	161	193	Á	225	á
2	34	"	B	98	b	130	162	194	Â	226	â
3	35	#	C	99	c	131	163	195	Ã	227	ã
4	36	\$	D	100	d	132	164	196	Ä	228	ä
5	37	%	E	101	e	133	165	197	Å	229	å
6	38	&	F	102	f	134	166	198	Æ	230	æ
7	39	'	G	103	g	135	167	199	Ç	231	ç
8	40	(H	104	h	136	168	200	È	232	è
9	41)	I	105	i	137	169	201	É	233	é
10	42	*	J	106	j	138	170	202	Ê	234	ê
11	43	+	K	107	k	139	171	203	Ë	235	ë
12	44	,	L	108	l	140	172	204	Ì	236	ì
13	45	-	M	109	m	141	173	205	Í	237	í
14	46	.	N	110	n	142	174	206	Î	238	î
15	47	/	O	111	o	143	175	207	Ï	239	ï
16	48	0	P	112	p	144	176	208	Ð	240	ð
17	49	1	Q	113	q	145	177	209	Ñ	241	ñ
18	50	2	R	114	r	146	178	210	Ò	242	ò
19	51	3	S	115	s	147	179	211	Ó	243	ó
20	52	4	T	116	t	148	180	212	Ô	244	ô
21	53	5	U	117	u	149	181	213	Õ	245	õ
22	54	6	V	118	v	150	182	214	Ö	246	ö
23	55	7	W	119	w	151	183	215	×	247	÷
24	56	8	X	120	x	152	184	216	Ø	248	ø
25	57	9	Y	121	y	153	185	217	Ù	249	ù
26	58	:	Z	122	z	154	186	218	Ú	250	ú
27	59	;	[123	{	155	187	219	Û	251	û
28	60	<	\	124		156	188	220	Ü	252	ü
29	61	=]	125	}	157	189	221	Ý	253	ý
30	62	>	^	126	~	158	190	222	Þ	254	þ
31	63	?	_	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): pueden utilizarse dos bytes, cuatro, 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 (decimal) x, se pulsa ALT 0x. Por ejemplo, para mostrar el carácter asociado al codepoint 35, hay que pulsar ALT 035

Aún más, 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.

Para simplificar, 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.5.4.2. Tratamiento de caracteres en C++

En el código fuente de un programa de C++, podemos asignar a cualquier **entero** un carácter de la plataforma, siempre que lo encerremos entre comillas simples. El valor que el entero almacena es el número de orden de dicho carácter en la tabla.

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

En lo que sigue, supondremos que la plataforma en la que se está escribiendo y ejecutando el programa usa un super-conjunto de la tabla definida en ISO-8859-1. Así pues, también podemos mostrar la ñ en el código:

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

Pero lo siguiente no lo es:

```
int letra_piso;  
  
letra_piso = ñ; // Error de compilación  
letra_piso = "ñ"; // Error de compilación  
letra_piso = 'ñ'; // Error de compilación
```

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.
- ▷ El tipo entero y los caracteres no se comportan adecuadamente con los operadores de E/S `cin` y `cout`.

```
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.

Tradicionalmente, el tipo de dato `char` es el más usado en C++ para trabajar con caracteres:

- ▷ El tipo `char` suele ser (ver comentarios después) un tipo 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
```

Nota:

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.

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.

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 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.5.4.3. Literales de carácter

Son tokens formados por:

- ▷ Un único carácter encerrado entre comillas simples:

'!' 'A' 'a' '5' 'ñ'

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

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

Observad la diferencia:

```
double r;
char letra;

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

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

Secuencia	Significado
\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

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

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

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

Escribiría en pantalla:

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

Esto es una comilla simple: '

Ampliación:

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

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

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



I.5.4.4. Funciones estándar y operadores

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

```

tolower    toupper

#include <cctype>
using namespace std;

int main(){
    char letra_piso;

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

```

Los operadores aplicables a los enteros también son aplicables a cualquier `char` o a cualquier literal de carácter. El operador actúa siempre sobre el entero de orden correspondiente:

```

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

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

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

```

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

I.5.5. El tipo de dato cadena de caracteres

Una cadena de caracteres nos permite guardar y manipular una *sucesión de caracteres* (una palabra o una serie de palabras). Cada componente o elemento de una cadena de caracteres es un dato de tipo `char`.

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

1. Las *cadenas estilo C* (también llamadas *cadenas clásicas*) son *vectores* -colección de datos del mismo tipo que se almacenan en posiciones consecutivas de memoria- de caracteres. Hay un carácter especial que indica el final y delimita la cadena: el carácter nulo '`\0`'.

Se verá en la asignatura Metodología de la Programación.

2. Usando el tipo `string` (la recomendada en esta asignatura)

El tipo string no pertenece al conjunto de tipos predefinidos de C++, sino que se trata de una clase externa.

Para poder operar con un dato string debemos incluir el paquete de recursos string:

```
#include <string>
```

Nota:

Formalmente, el tipo `string` no es un tipo simple, sino compuesto (está formado por varios caracteres). No obstante, lo incluimos dentro de este tema en aras de simplificar la materia.

I.5.5.1. Literales de cadena de caracteres

Un literal de tipo **cadena de caracteres** se especifica escribiendo una serie de caracteres entre comillas dobles. Por ejemplo, "Hola" y "a" son literales de cadena de caracteres.

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

Las **secuencias de escape** (caracteres especiales indicados en la sección I.5.4.3) también pueden aparecer de manera explícita en un literal de cadena de caracteres (no dejan de ser otros caracteres):

```
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 \"";
```

Su ejecución produce como resultado:

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

Esto es una comilla simple ’ y esto es una comilla doble "
```

Ejercicio. Determinar cuales de las siguientes son cadenas de caracteres válidas, y determinar qué se mostraría si se enviase 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"

Nota:

Un literal de cadena de caracteres no es formalmente un **dato string** (es una cadena clásica constante) aunque puede emplearse para dar valor a un **dato string** y como argumento en las operaciones que esperan como operando un **dato string** (en ese caso se convierten de manera *automática* a un **dato string**).

I.5.5.2. Operaciones con datos string

Muy importante: Al declarar un **dato de tipo string** contiene, por defecto, la **cadena vacía (empty string)**, a saber "" (su *longitud* o número de caracteres que la componen es 0).

Las primeras operaciones que realizamos sobre **datos string** son:

▷ Asignar un valor concreto al objeto **string** a través de un **literal de cadena de caracteres**.

▷ Enviar su contenido a **cout** con el operador **<<**

Un ejemplo que crea un **string** vacío, lo inicializa usando un **literal de cadena de caracteres** y envía su contenido a **cout** es:

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

int main(){
    string mensaje_bienvenida; // cadena vacía
    mensaje_bienvenida = ";Hola, mundo!"; // Se usa un literal
    cout << mensaje_bienvenida << endl;
}
```

Su ejecución da como resultado:

;Hola, mundo!

- ▷ **Concatenar dos datos string o un string con un carácter o con un literal de cadena de caracteres usando el operador +**

```
string cad; // cadena vacía
cad = "Hola y ";
cad = cad + "adiós";
cout << cad << endl;
```

Su ejecución da como resultado:

Hola y adiós

Observe que los caracteres de especiales '\t', '\n', ... pueden ser parte de una cadena:

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

int main(){
    string cadena; // ""

    cadena = "\tFunda"; // "\tFunda"
    cadena = cadena + "ment"; // "\tFundament"
    cadena = cadena + 'o'; // "\tFundamento"
    cadena = cadena + "s\n"; // "\tFundamentos\n"

    cout << cadena << endl;
}
```

Su ejecución da como resultado:

Fundamentos

La clase `string` proporciona un conjunto muy amplio de operaciones. Además de las indicadas anteriormente en esta asignatura emplearemos las siguientes:

`push_back()` Añade un carácter a un `string`
`length()` Devuelve el número de caracteres de un `string`
`clear()` “Limpia” el `string` (lo deja vacío)
`at()` Para acceder a las componentes individuales de un `string`
 (a cada uno de los caracteres que la componen)

...

```
string cad;

cad.push_back ('H');
cad.push_back ('o');
cad.push_back ('l');
cad.push_back ('a');

cout << cad << endl; // Hola
cout << "longitud = " << cad.length() << endl; // 4
cout << "caracter pos.1 --> " << cad.at(1) << endl; // o

cad.clear(); // "Limpia" el string
cout << cad << endl; // (vacía --> No muestra nada)
cout << "longitud = " << cad.length() << endl; // 0
```

Su ejecución da como resultado:

```
Hola
longitud = 4
caracter pos.1 --> o
```

0

I.5.5.3. Operaciones de conversión a string / desde string

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
}
```

I.5.5.4. Lectura de datos string: la función getline

Si el valor que vamos a proporcionar tiene una sola palabra podríamos usar el operador `>>` con `cin`, igual como se ha leído el valor de los datos de tipos simples hasta ahora.

Recuerde que el operador `>>` lee una única palabra (texto) y su contenido lo transforma al tipo de dato que aparece a su derecha.

Si el texto introducido tiene más de una palabra no se debe usar el operador `>>` con `cin` porque leerá únicamente la primera palabra:

```
string nombre;

cout << "Nombre y apellidos: ";
cin >> nombre;

cout << nombre;
```

Una posible ejecución será:

```
Nombre y apellidos: Francisco Jose
Francisco
```

Tenemos, además, un problema añadido, y es que Jose se queda en el buffer de entrada, y la siguiente lectura que hiciera el programa tomaría ese valor.

La lectura de string se realizará con la función getline. La función necesita dos argumentos: el flujo desde donde se toman los datos (cin en nuestro caso) y el dato string donde se van a copiar:

```
string nombre;

cout << "Nombre y apellidos: ";
getline (cin, nombre);

cout << nombre;
```

Una posible ejecución será:

```
Nombre y apellidos: Francisco Jose
Francisco Jose
```

I.5.6. El tipo de dato lógico o booleano

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

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

I.5.6.1. Rango. Representación

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

Un dato `bool` no guarda estos valores, sino que se usa un valor entero (ojo, no `int`). El estándar simplemente dice al respecto que el tamaño no puede ser menor a un byte. Lo usual es emplear una representación entera de un byte.

C++ considera que `false` es equivalente a 0 y todo lo que no sea cero es `true`.

La asimilación del valor 0 con la etiqueta `false` tendrá consecuencias muy importantes en nuestros programas, como iremos descubriendo.

Ejemplo. Observe el resultado de este código:

```
bool trabajo = true;
bool descanso = false;
cout << "trabajo vale " << trabajo << endl;
cout << "descanso vale " << descanso << endl;
cout << "tamaño (en bytes) = " << sizeof(bool) << endl;
```

Los valores mostrados de los datos `bool` son numéricos:

```
trabajo vale 1  
descanso vale 0  
tamaño (en bytes) = 1
```

Podemos, no obstante, indicar que a la hora de visualizar el valor de un dato `bool` se muestre su etiqueta más que su valor numérico con el manipulador `boolalpha`:

```
cout << "trabajo vale " << boolalpha << trabajo << endl;  
cout << "descanso vale " << boolalpha << descanso << endl;
```

Ahora, el resultado sería:

```
trabajo vale true  
descanso vale false
```

I.5.6.2. 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`)

```
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_indefinido;

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

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

$p \equiv \text{Carlos es varón}$

$q \equiv \text{Carlos es joven}$

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`. Asígnaleles cualquier valor. Declare otra variable `es_varon_viejo` y asígnale el valor correcto usando las variables anteriores y los operadores lógicos.

I.5.6.3. Operadores Relacionales

Son los operadores habituales de comparación de expresiones numéricas.

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

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

Algunos ejemplos:

- ▷ La expresión `(4 < 5)` devuelve valor `true`
- ▷ La expresión `(4 > 5)` devuelve el valor `false`
- ▷ La relación de orden entre caracteres se establece según la tabla ASCII.
La expresión `('a' > 'b')` devuelve el valor `false`.

`!=` es el operador relacional distinto.

`!` es la negación lógica.

`==` es el operador relacional de igualdad.

`=` es la operación de asignación.

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

```
// Ejemplo de operadores relacionales

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

    entero1 = 3;
    entero2 = 5;

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

    real1 = 3.8;
    real2 = 8.1;

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

Veremos su uso en la sentencia condicional:

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

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

Tabla de Precedencia:

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

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

```

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

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

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

```



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



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

I.5.7. Lectura de varios datos

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

```
int entero, otro_entero;
double real;

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

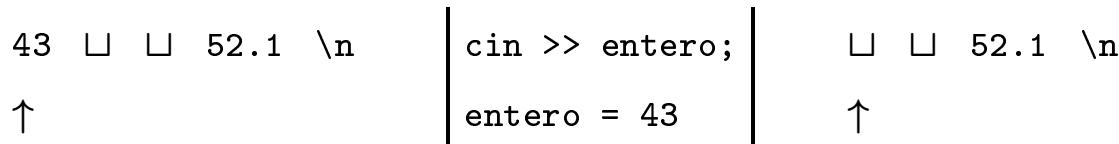
También podríamos haber usado como separador un espacio en blanco o un tabulador. ¿Cómo funciona?

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

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

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

Supongamos que el usuario introduce 43 52.1<Enter>



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

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

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

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

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

cin >> real;
// real = 52.1
// Buffer: [\n]

cin >> otro_entero;
// Buffer: []
```

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

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

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

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

Contenido de fichero.txt:

```
43      52.1\n37
```

Desde un editor de texto se vería lo siguiente:

```
43      52.1  
37
```

La lectura sería así:

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

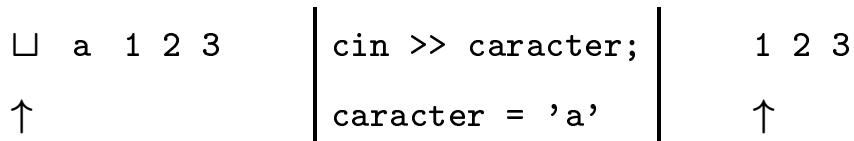
¿Qué pasa si queremos leer un entero pero introducimos, por ejemplo, una letra? Se produce un error en la lectura y a partir de ese momento, todas las operaciones siguientes de lectura también dan fallo y el cursor no avanzaría.

□ □ a □ 1 2 3 ↑	cin >> entero; Fallo de lectura cin >> lo_que_sea; Fallo de lectura	□ □ a □ 1 2 3 ↑
--------------------	--	--------------------

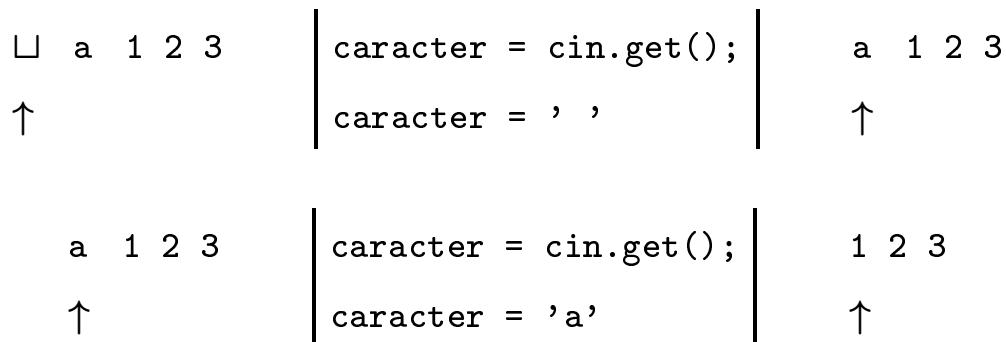
Se puede resetear el estado de la lectura con `cin.clear()` y consultarse el estado actual (error o correcto) con `cin.fail()`. En cualquier caso, para simplificar, a lo largo de este curso asumiremos que los datos vienen en el orden correcto especificado en el programa, por lo que no será necesario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

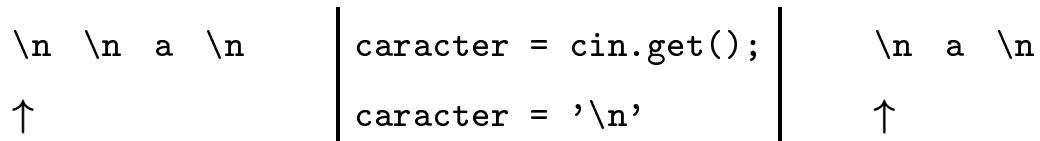
```
char caracter;
```



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



Lo mismo ocurre si hubiese un carácter de nueva línea:



Ampliación:

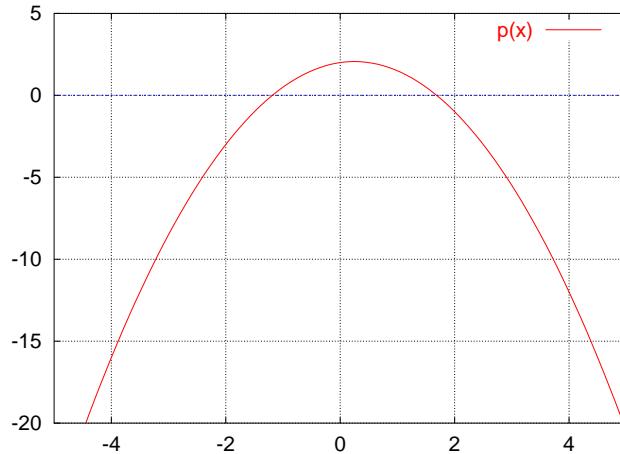


Para leer una cadena de caracteres (`string`) podemos usar la función `getline` de la biblioteca `string`. Permite leer caracteres hasta llegar a un terminador que, por defecto, es el carácter de nueva línea '`\n`'. La cadena a leer se pasa como un parámetro por referencia a la función. Este tipo de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

I.6. El principio de una única vez

Ejemplo. 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 `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 cambiar dos líneas.



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

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

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

    // Cada expresión sólo se evalúa una vez:
    denominador = 2*a;           
    radical = sqrt(b*b - 4*a*c);

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

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

Nota:

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

Principio de Programación:

Una única vez (Once and only once)



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

O dicho de una manera informal:

Jamás ha de repetirse código

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

En el tema III (Funciones) 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.

Tema II

Estructuras de control

Objetivos:

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

Autor: Juan Carlos Cubero.

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

II.1. Estructura condicional

II.1.1. Flujo de control

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

Una forma de especificarlo es numerando las líneas de un programa. Por ejemplo, recuerde el ejemplo de la página 114 (no numeramos las sentencias de declaración de los datos):

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

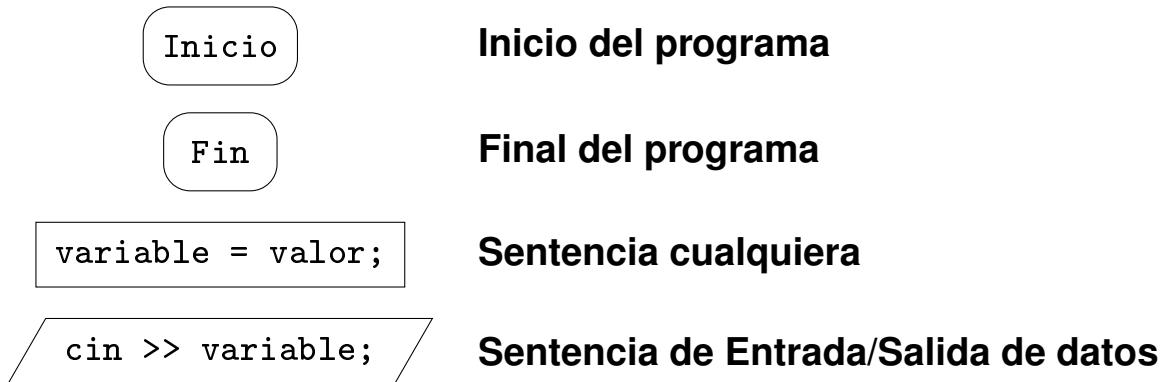
1   cout << "\nIntroduce coeficiente de 2º grado: ";
2   cin >> a;
3   cout << "\nIntroduce coeficiente de 1er grado: ";
4   cin >> b;
5   cout << "\nIntroduce coeficiente independiente: ";
6   cin >> c;

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

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

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

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



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

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

▷ **Hacia delante.**

Implicará que un conjunto de sentencias no se ejecutarán.

Vienen definidos a través de una estructura condicional.

▷ **Hacia atrás.**

Implicará que volverá a ejecutarse un conjunto de sentencias.

Vienen definidos a través de una estructura repetitiva.

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

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

Existen tres tipos: *Simple, Doble y Múltiple*

II.1.2. Estructura condicional simple

II.1.2.1. Formato

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

<condición> es una expresión lógica

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

Los paréntesis que encierran la condición son obligatorios.

Todo el bloque que empieza por if y termina con la última sentencia 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 118

```
#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;

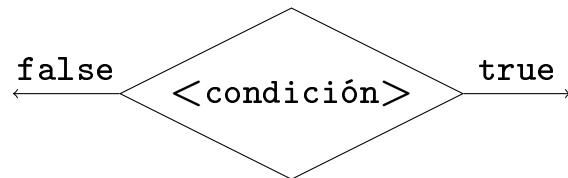
        cout << "\nLas raíces son" << raiz1 << " y " << raiz2;
    }
}
```



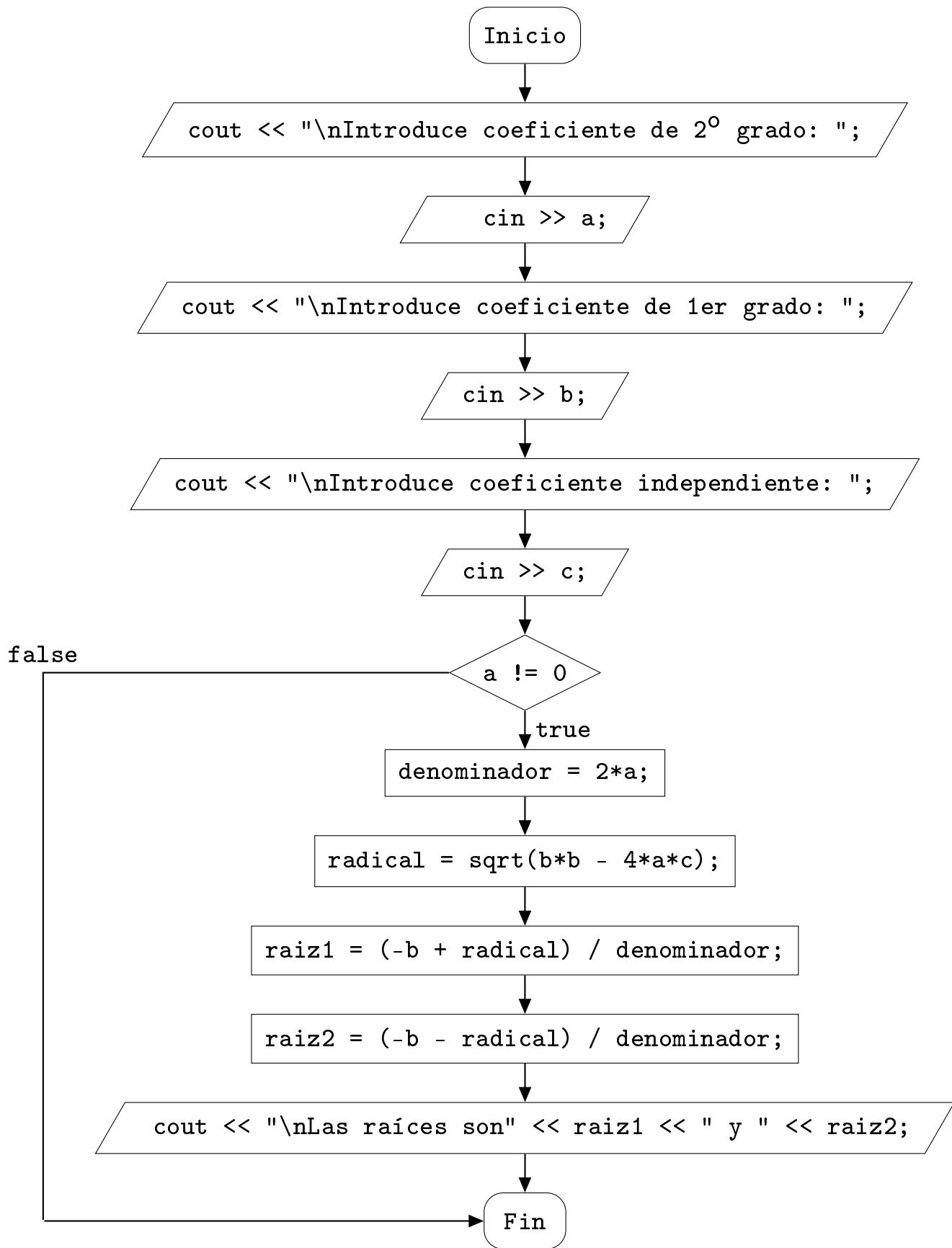
208

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
```

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;
```

Ejemplo. En un programa de gestión de notas, suba medio punto a los que han sacado más de 4.5 puntos en el examen escrito.

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

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

II.1.2.3. Cuestión de estilo

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

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

```
cin >> c;  
                    // <- 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

No seguir estas normas baja puntos en el examen.

II.1.2.4. Condiciones compuestas

En muchos casos tendremos que utilizar condiciones compuestas:

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";
```



142

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 < 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 < 1300)  
    salario_final = salario_final * 1.04;
```

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

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

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

II.1.2.5. Variables no asignadas en los condicionales

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

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



Ejemplo. Retome el ejemplo del intervalo de la página 130 y asígnele el valor correcto a una variable de tipo `bool` `pertenece_al_intervalo`:

```
double inferior, superior, dato;  
bool pertenece_al_intervalo;  
  
cout << "\nIntroduzca los extremos del intervalo: ";  
cin >> inferior;  
cin >> superior;  
cout << "\nIntroduzca un real arbitrario: ";  
cin >> dato;  
  
pertenece_al_intervalo = false;  
  
if (dato >= inferior && dato <= superior)  
    pertenece_al_intervalo = true;
```

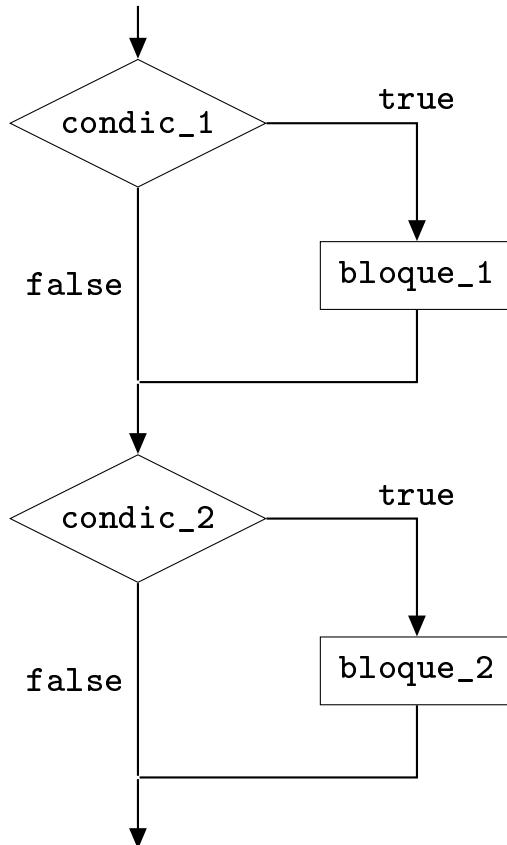


Debemos prestar especial atención a los condicionales en los que se asigna un primer valor a alguna variable. Intentaremos garantizar que dicha variable salga siempre del condicional (independientemente de si la condición era verdadera o falsa) con un valor establecido.

II.1.2.6. Estructuras condicionales consecutivas

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

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



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

```
int edad, altura;  
cin >> edad;  
cin >> altura;  
  
if (edad >= 18)  
    cout << "\nEs mayor de edad";  
  
if (altura >= 190)  
    cout << "\nEs alto/a";
```



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

Ejemplo. Retome el ejemplo de la subida salarial. Si la persona tiene más de dos hijos, súbale un 2% adicional. Esta subida es aplicable a todo el mundo y se realizará sobre el sueldo ya incrementado con anterioridad (en su caso) el 4%.

```
.....  
salario_final = salario_base;  
  
if (edad >= 45 && salario < 1300)  
    salario_final = salario_final * 1.04;  
  
if (numero_hijos > 2)  
    salario_final = salario_final * 1.02;
```

Ejemplo. En el ejemplo de la subida de la nota de la página 128, controle que la nota obtenida no sea mayor de 10.

```
double nota_escrito;

cin >> nota_escrito;

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

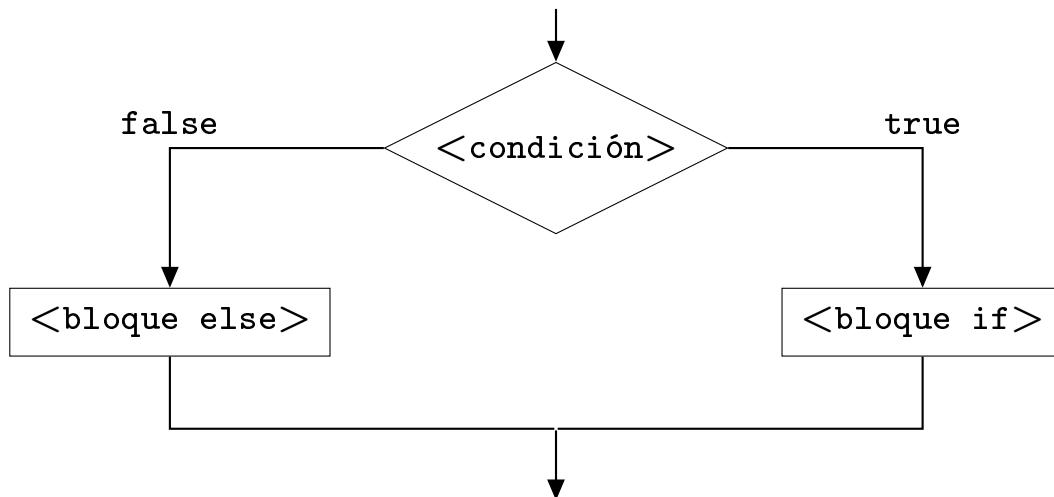
if (nota_escrito > 10)
    nota_escrito = 10;
```

II.1.3. Estructura condicional doble

II.1.3.1. Formato

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

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



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

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

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

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

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

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

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

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



Ejemplo. Compruebe si un número es par (continuación)

```
cin >> entero;

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

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

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

Ejemplo. Compruebe si el salario es bajo (continuación)

```
double salario;

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

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

Ejemplo. En el ejemplo de la página 128 se subía la nota a los que tenían más de 4.5. ¿Qué ocurre si la nota es menor?

```
cin >> nota_escrito;

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



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

En definitiva, en este ejemplo, no es necesaria una estructura condicional doble sino sencilla, por lo que nos quedamos con la versión de la página 128.

Ejemplo. Máximo de dos valores (continuación de lo visto en la página 132).

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

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

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

Ambas formas son correctas.

Ejemplo. Valor dentro de un intervalo (continuación de lo visto en la página 142)

```
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`

```
pertenece_al_intervalo = false;  
  
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:

```
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

Ejemplo. El ejemplo del número par podría haberse resuelto usando dos condicionales simples consecutivos, en vez de un condicional doble:

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

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

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



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

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

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

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

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



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

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

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



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

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

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

Ejemplo.

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

```
if (entero % 2 == 0)      if (edad >= 18)      if (a != 0)
```

.....

```
if (entero % 2 != 0)      if (edad < 18)      if (a == 0)
```

.....



```
if (entero % 2 == 0)      if (edad >= 18)      if (a != 0)
```

.....

```
else                      else                     else
```

.....



Cuando trabajamos con expresiones compuestas, se hace aún más patente la necesidad de usar un condicional doble:

Ejemplo. Retome el ejemplo de la subida salarial de la página 135. Si no se cumple la condición de la subida del 4%, el salario únicamente se incrementará en un 1%.

```
edad >= 45 && salario < 1300 -> +4%
```

Otro caso -> +1%

.....

```
salario_final = salario_base;
```

```
if (edad >= 45 && salario < 1300)
    salario_final = salario_final * 1.04;
else
    salario_final = salario_final * 1.01;
```



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

```
edad >= 45 && salario < 1300 -> +4%
```

edad < 45 || salario >= 1300 -> +1%

El código siguiente daría un resultado correcto pero repite código por lo que siempre usaremos la estructura condicional doble:

```
salario_final = salario_base;
```

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

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



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.

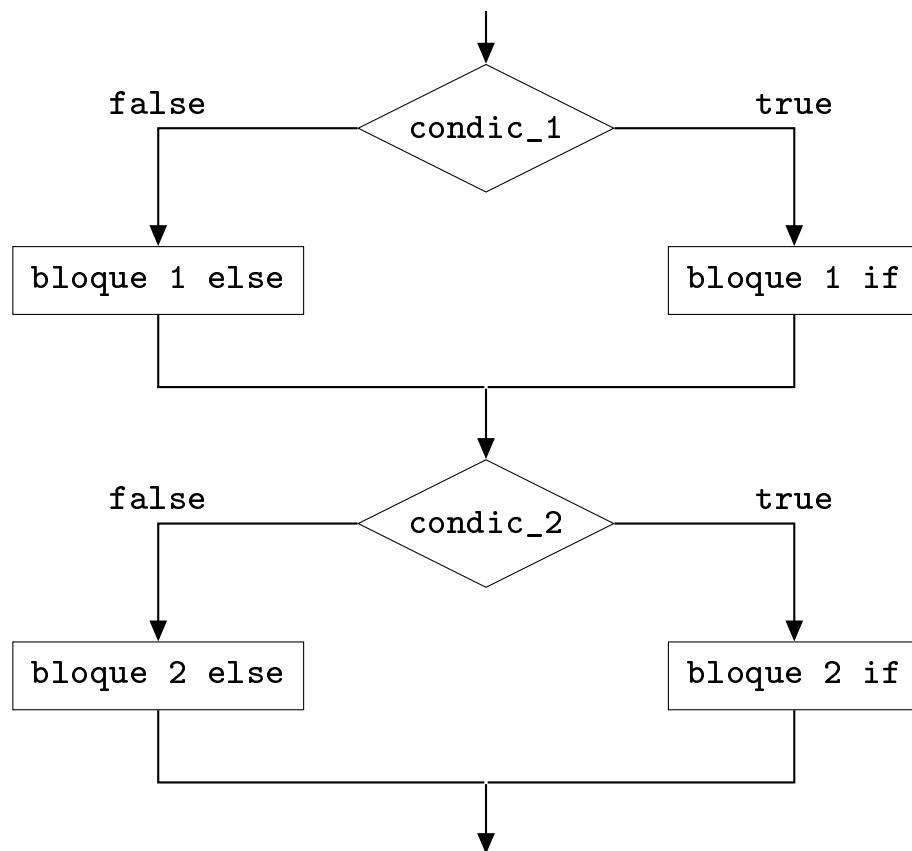
```
if (cond)           if (cond)  
...                 ...  
if (!cond)         else  
...                 ...
```

II.1.3.3. Estructuras condicionales dobles consecutivas

Un condicional doble garantiza la ejecución de un bloque de instrucciones: o bien el bloque del `if`, o bien el bloque del `else`. Si hay dos condicionales dobles consecutivos, se ejecutarán los bloques de instrucciones que correspondan: o ninguno, o uno de ellos, o los dos.

```
if (condic_1)
    bloque 1 if
else
    bloque 1 else
```

```
if (condic_2)
    bloque 2 if
else
    bloque 2 else
```



Veamos un ejemplo de un condicional doble y otro simple consecutivos.

Ejemplo. Retome el ejemplo de la subida salarial de la página 147. Añada la subida salarial por número de hijos: si tiene más de dos hijos, se subirá un 2% adicional sobre el salario incrementado anteriormente.

```
edad >= 45 && salario < 1300 -> +4%
```

```
Otro caso -> +1%
```

```
numero_hijos > 2 -> +2%
```

Nos quedaría:

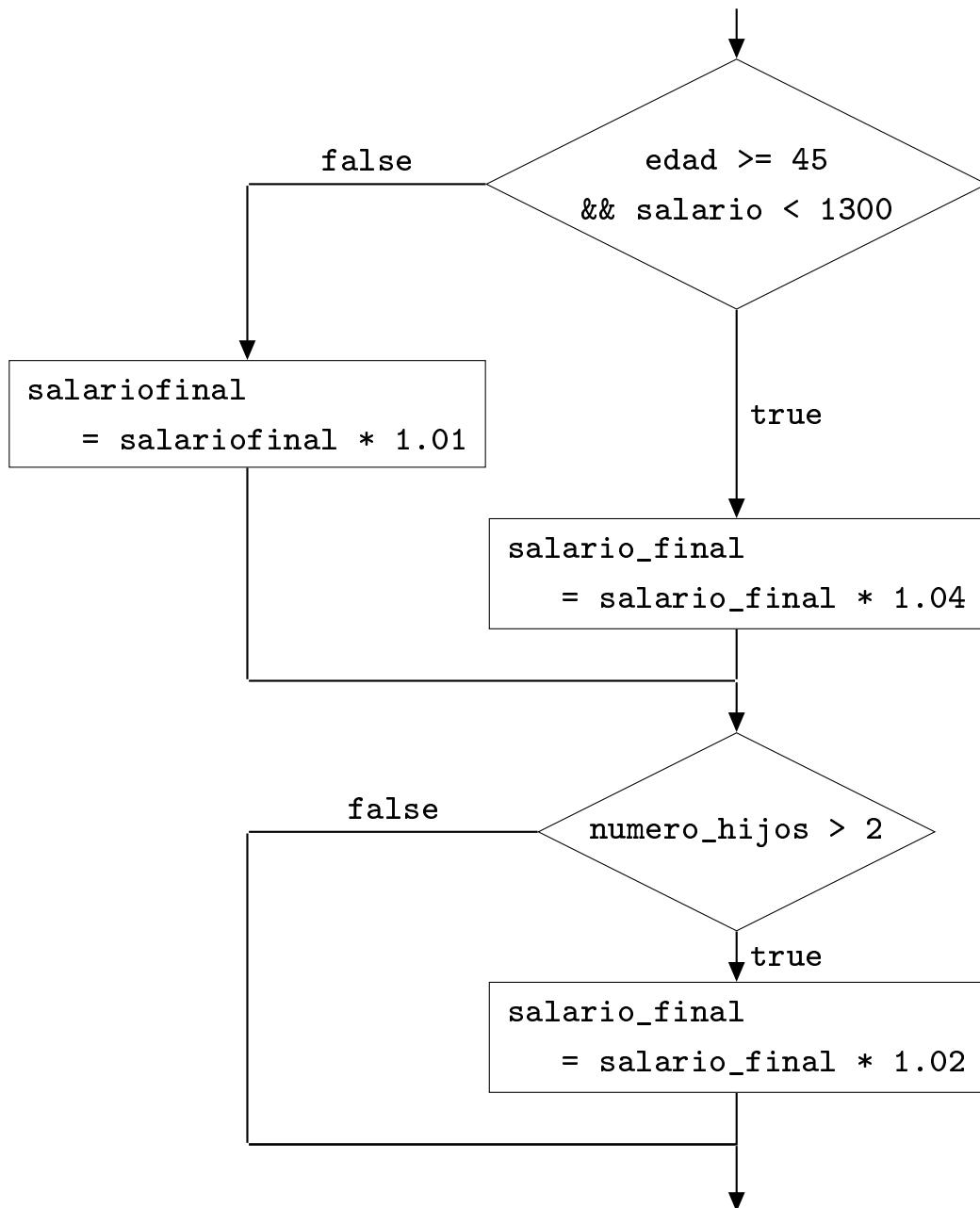
```
.....
```

```
salario_final = salario_base;
```

```
if (edad >= 45 && salario < 1300)
    salario_final = salario_final * 1.04;
else
    salario_final = salario_final * 1.01;
```

```
if (numero_hijos > 2)
    salario_final = salario_final * 1.02;
```

Recordemos que la segunda subida se realiza sobre el salario final ya modificado en el primer condicional. En prácticas se propone realizar las subidas 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.

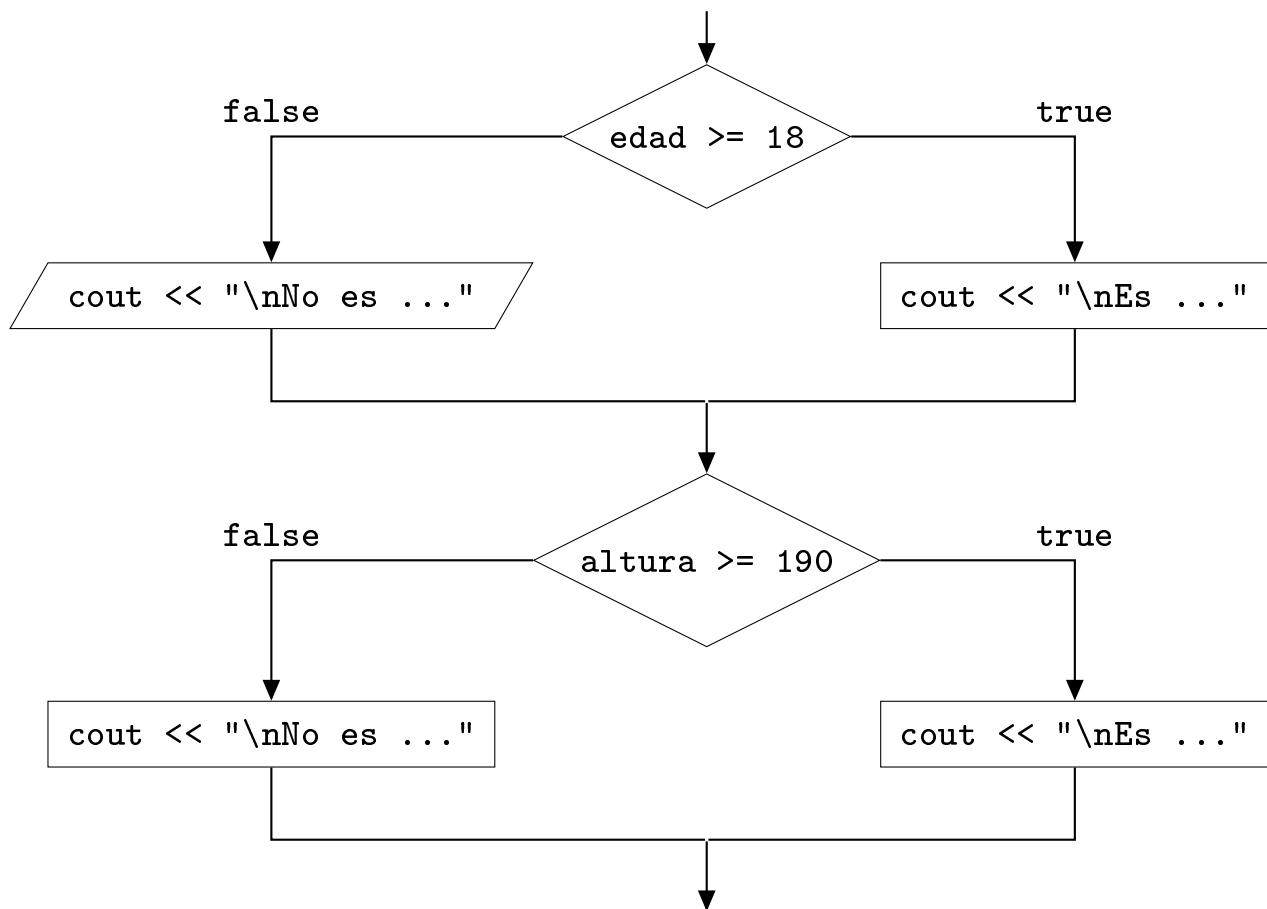
```
int edad, altura;  
  
cin >> edad;  
cin >> altura;  
  
if (edad >= 18)  
    cout << "\nEs mayor de edad";  
else  
    cout << "\nEs menor de edad";  
  
if (altura >= 190)  
    cout << "\nEs alto/a";  
else  
    cout << "\nNo es alto/a";
```



203

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 2ⁿ situaciones posibles.



Tal y como veíamos en la página 144, si hubiésemos duplicado las expresiones de los condicionales, 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";
```

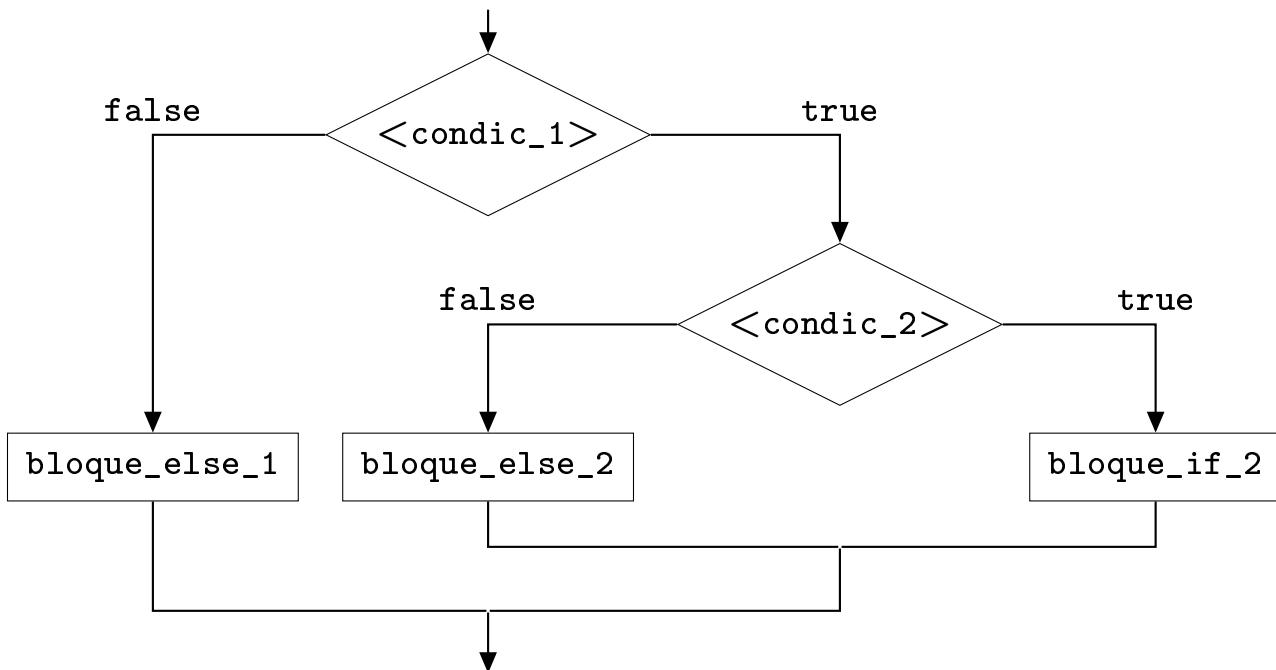


II.1.4. Anidamiento de estructuras condicionales

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_5	false	da igual	true

Ejemplo. Retomamos el ejemplo de subir la nota de la página 136:

```
double nota_escrito;

cin >> nota_escrito;

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

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

Aunque la salida de este algoritmo sea correcta, realmente, sólo es necesario truncar a diez cuando se ha subido la nota. Por lo tanto, mejor si usamos un condicional anidado:

```
double nota_escrito;

cin >> nota_escrito;

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

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

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

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."  
}
```



208

Ejemplo. Modificamos el ejemplo de la edad y la altura de una persona visto en la página 152. Si es mayor de edad, decimos si es alto o no. Si es menor de edad no hacemos nada más.

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

```
int edad, altura;  
  
cin >> edad;  
cin >> altura;  
  
if (edad >= 18){  
    cout << "\nEs mayor de edad";  
  
    if (altura >= 190)  
        cout << "\nEs alto/a";  
    else  
        cout << "\nNo es alto/a";  
}  
else // <- edad < 18  
    cout << "\nEs menor de edad";
```



203

Ejemplo. Cambiamos el ejemplo anterior para implementar el siguiente criterio: Si es mayor de edad, el umbral para decidir si una persona es alta o no es 190 cm. Si es menor de edad, el umbral baja a 175 cm.

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

```
int edad, altura;  
  
cin >> edad;  
cin >> altura;  
  
if (edad >= 18){  
    cout << "\nEs mayor de edad";  
  
    if (altura >= 190)  
        cout << "\nEs alto/a";  
    else  
        cout << "\nNo es alto/a";  
}  
else{  
    // <- edad < 18  
    cout << "\nEs menor de edad";  
  
    if (altura >= 175)  
        cout << "\nEs alto/a";  
    else  
        cout << "\nNo es alto/a";  
}
```



203

Observe que si el umbral de la altura fuese el mismo para cualquier edad, la solución es la que vimos en la página 152.

Ejemplo. Retomemos el ejemplo de la subida salarial de la página 150. La subida por número de hijos se aplicaba a todo el mundo:

```
edad >= 45 && salario < 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 < 1300    -> +4%
```

```
    numero_hijos > 2            -> +2%
```

```
Otro caso                      -> +1%
```

Nos quedaría:

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

if (edad >= 45 && salario < 1300){
    salario_final = salario_final * 1.04;

    if (numero_hijos > 2)
        salario_final = salario_final * 1.02;
}

else
    salario_final = salario_final * 1.01;
```

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 o un salario por debajo de 1300. En otro caso se sube un 1%

```
edad >= 45 && salario < 1300 -> +4%
En otro caso -> +1%
```

.....

```
salario_final = salario_base;
```

```
if (edad >= 45 && salario < 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 < 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:

```
if (c1 && c2)           if (c1)
    bloque_A             if (c2)
else                   bloque_A
    bloque_B             else
                        bloque_B
                        else
                        bloque_B
```

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 < 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 < 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;
```

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 < 1300)  
    salario_final = salario_final * 1.04;  
if (experiencia > 2 && edad >= 45 && salario < 1300 && numero_hijos > 2)  
    salario_final = salario_final * 1.02;  
if (experiencia > 2 && (edad < 45 || salario > 1300))  
    salario_final = salario_final * 1.01;  
if (experiencia <= 2)  
    salario_final = salario_final * 0.99;
```



Obviamente, este código es nefasto ya que no cumple el principio de una única vez.

```
if (c1)  
    if (c1 && c2 && c3)  
        <accion_A>  
    if (c1 && c2 && !c3)  
        <accion_B>  
    if (c1 && !c2)  
        <accion_C>  
    if (!c1)  
        <accion_D>  
    if (c2)  
        if (c3)  
            <accion_A>  
        else  
            <accion_B>  
        else  
            <accion_C>  
    else  
        <accion_D>
```



Podemos resumir lo visto en el siguiente consejo (que es una generalización del visto en la página 148)

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. Menú de operaciones.

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

int main(){
    double dato1, dato2, resultado;
    char opcion;

    cout << "\nIntroduce el primer operando: ";
    cin >> dato1;
    cout << "\nIntroduce el segundo operando: ";
    cin >> dato2;
    cout << "\nElija (S)Sumar, (R)Restar, (M)Multiplicar: ";
    cin >> opcion;
    opcion = toupper(opcion);

    if (opcion == 'S')
        resultado = dato1 + dato2;
    if (opcion == 'R')
        resultado = dato1 - dato2;
    if (opcion == 'M')
        resultado = dato1 * dato2;
    if (opcion != 'R' && opcion != 'S' && opcion != 'M')
        resultado = NAN; // <- Hay que incluir cmath

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



168

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. No es un problema de código duplicado sino de eficiencia.

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:

```
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
}
```



También podríamos haber asociado las opciones a los caracteres '+', '-' y '*'.

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)   →     ...
...
if (c3)           else if (c2)
...
...
...
else if (c3)
```

II.1.5. Estructura condicional múltiple

Recuerde el ejemplo de lectura de dos enteros y una opción de la página 168. 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.
- ▷ <constante1> ó <constante2> es un literal entero.
- ▷ switch sólo comprueba la igualdad.
- ▷ No debe haber dos cases con la misma <constante> en el mismo switch. Si esto ocurre, sólo se ejecutan las sentencias del caso que aparezca primero.
- ▷ El identificador especial default permite incluir un caso por defecto, que se ejecutará si no se cumple ningún otro. Se suele colocar como el último de los casos.

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

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

II.1.6. Algunas cuestiones sobre condicionales

II.1.6.1. Álgebra de Boole

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

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

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

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

Ampliación:

Consultad:

http://es.wikipedia.org/wiki/Algebra_booleana

<http://serbal.pntic.mec.es/~cmunoz11/bole.pdf>



Ejemplo. Cambie el ejercicio de la subida salarial para que ésta se produzca si el empleado 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 || (salario < 45 && experiencia > 3))
    salario_final = salario_final * 1.04;
```

La expresión lógica $A \mid\mid (\neg A \ \&\& B)$ es equivalente a $A \mid\mid B$. Basta aplicar la ley distributiva:

$A \mid\mid (\neg A \ \&\& B) \leftrightarrow (A \mid\mid \neg A) \ \&\& (A \mid\mid B) \leftrightarrow \text{True} \ \&\& (A \mid\mid B) \leftrightarrow A \mid\mid B$

Aplicándolo a nuestro ejemplo, nos quedaría finalmente:

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

if (edad >= 45 || experiencia > 3)
    salario_final = salario_final * 1.04;
```

La expresión $A \mid\mid (\neg A \ \&\& B)$ la sustituiremos por la equivalente a ella: $A \mid\mid B$

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)
```

Obviamente, resulta mucho más intuitiva la siguiente expresión lógica:

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

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.



II.1.6.2. Cuidado con la comparación entre reales

La expresión `1.0 == (1.0/3.0)*3.0` podría evaluarse a `false` debido a la precisión finita para calcular $(1.0/3.0)$ (`0.333333333`)

Otro ejemplo:

```
double raiz_de_dos;

raiz_de_dos = sqrt(2.0);

if (raiz_de_dos * raiz_de_dos != 2.0)    // Podría evaluarse a true
    cout << "¡Raíz de dos al cuadrado no es igual a dos!";
```

Otro ejemplo:

```
double descuento_base, porcentaje;

descuento_base = 0.1;
porcentaje = descuento_base * 100;

if (porcentaje == 10.0)    // Podría evaluarse a false :-(
    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.1.6.3. Evaluación en ciclo corto y en ciclo largo

Evaluación en ciclo corto (Short-circuit evaluation) : El compilador optimiza la evaluación de expresiones lógicas evaluando sus términos de izquierda a derecha hasta que sabe el resultado de la expresión completa (lo que significa que puede dejar términos sin evaluar). La mayoría de los compiladores realizan este tipo de evaluación por defecto.

Evaluación en ciclo largo (Eager evaluation) : El compilador evalúa todos los términos de la expresión lógica para conocer el resultado de la expresión completa.

Ejemplo.

```
if ( (n != 0) && (abs(1/n) < 0.001) ) ...
```

Supongamos 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.

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

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

II.1.7. Programando como profesionales

II.1.7.1. Diseño de algoritmos fácilmente extensibles

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

Algoritmo: Mayor de tres números. Versión 1

- ▷ **Entradas:** a, b y c
Salidas: El mayor entre a, b y c
- ▷ **Descripción:**

Si a es mayor que los otros, el mayor es a

Si b es mayor que los otros, el mayor es b

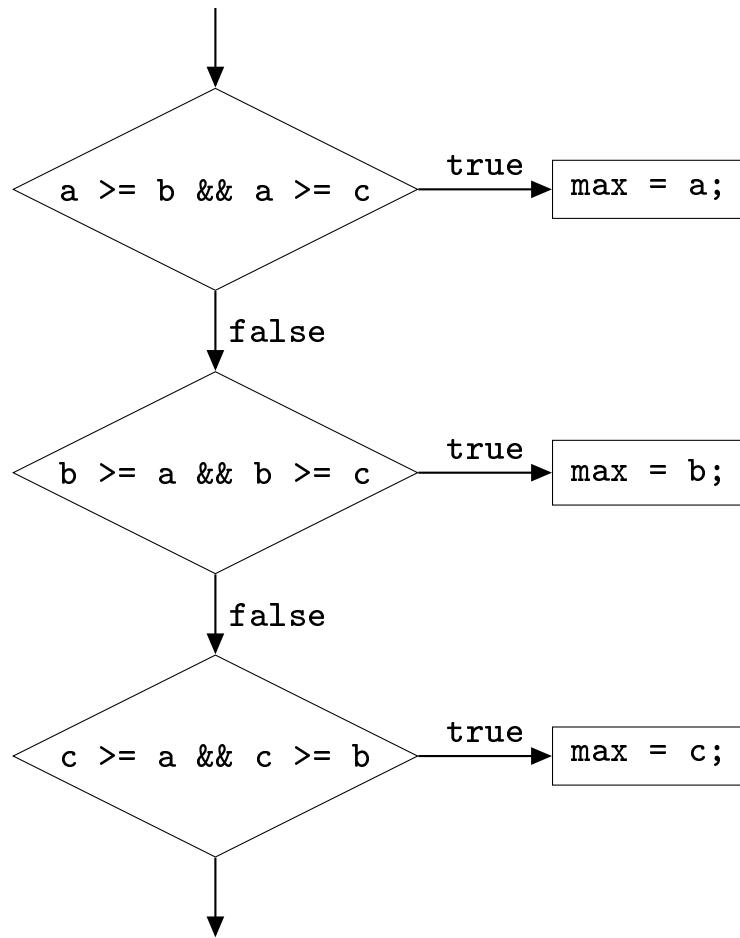
Si c es mayor que los otros, el mayor es c

- ▷ **Implementación:**

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



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 evaluaciones de expresiones lógicas.

Ejemplo. El mayor de tres números (segunda aproximación)

Algoritmo: Mayor de tres números. Versión 2

▷ **Entradas y Salidas:** idem

▷ **Descripción:**

Si a es mayor que b, entonces

Calcular el máximo entre a y c

En otro caso,

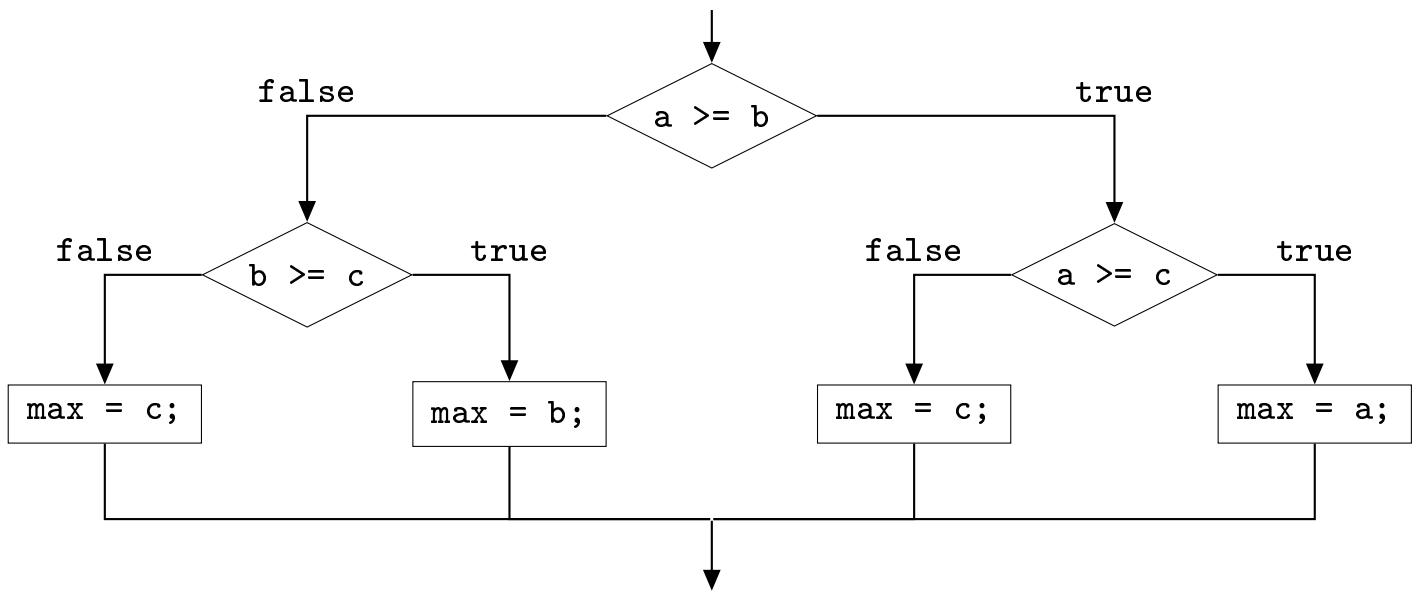
Calcular el máximo entre b y c

▷ **Implementación:**

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



187



Inconvenientes:

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

El mayor de cuatro números con la segunda aproximación:

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



187

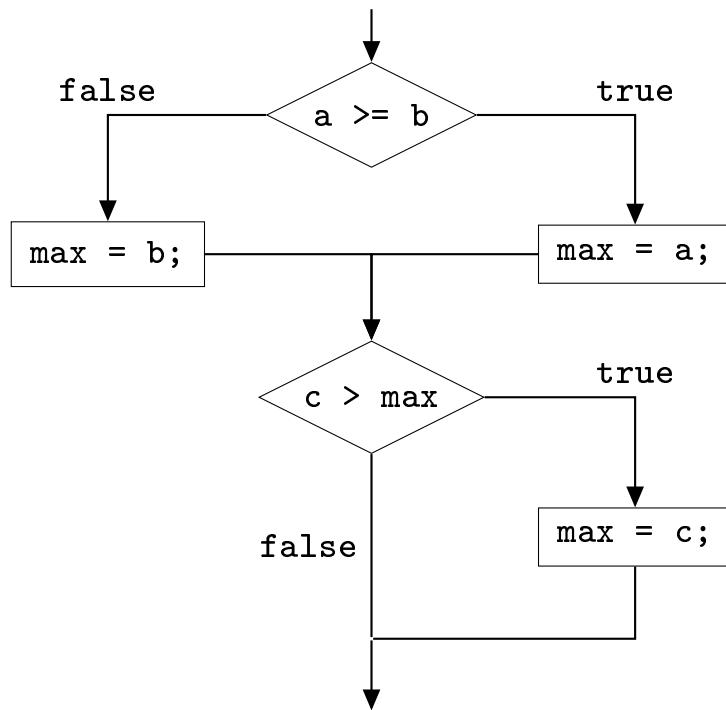
Ejemplo. El mayor de tres números (tercera aproximación)**Algoritmo:** Mayor de tres números. Versión 3

- ▷ **Entradas y Salidas:** idem
- ▷ **Descripción e Implementación:**

```
/*
Calcular el máximo (max) entre a y b.
Calcular el máximo entre max y c.
*/
if (a >= b)
    max = a;
else
    max = b;

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

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/~carlos/FP/II_max.cpp

En general:

Calcular el máximo (max) entre a y b.
Para cada uno de los valores restantes,
max = máximo entre max y dicho valor.

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

IMPORTANT

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.1.7.2. Descripción de un algoritmo

Se trata de describir la idea principal del algoritmo, de forma concisa y esquemática, sin entrar en detalles innecesarios.

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

```
else
```

```
    max = b;
```

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

Una lamentable descripción del algoritmo:

Compruebo si $a \geq b$. En ese caso, lo que hago es asignarle a la variable max el valor a y si no le asigno el otro valor, b. Una vez hecho esto, paso a comprobar si el otro valor, es decir, c, es mayor que max (la variable anterior), en cuyo caso le asigno a max el valor c y si no, no hago nada.



El colmo:

Compruebo si $a \geq b$ y en ese caso lo que ago es asignarle a la variable max el valor a y sino le asigno el otro valor b una vez echo esto paso a comprovar 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 (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.

No seguir estas normas baja puntos en el examen.

IMPORTANT

II.1.7.3. Descomposición de una solución en tareas más sencillas

Si observamos una misma operación en todas las partes de una estructura condicional, debemos ver si es posible su extracción fuera de la estructura.

Ejemplo. Retomemos el ejemplo de la subida salarial de la página 164.

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

if (experiencia > 2)  {
    if (edad >= 45 && salario < 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 apreciar que la actualización de `salario_final` siempre se hace de la misma forma. Podemos resolver el problema descomponiendo la tarea principal en dos sub-tareas:

1. Calcular el porcentaje de actualización
2. Aplicar dicho porcentaje

```
double porcentaje_actualizacion;  
.....  
salario_final = salario_base;  
  
if (experiencia > 2) {  
    if (edad >= 45 && salario < 1300){  
        porcentaje_actualizacion = 1.04;  
  
        if (numero_hijos > 2) // Observe esta asignación:  
            porcentaje_actualizacion = porcentaje_actualizacion * 1.02;  
    }  
    else  
        porcentaje_actualizacion = 1.01;  
}  
else  
    porcentaje_actualizacion = 0.99;  
  
salario_final = salario_final * porcentaje_actualizacion;
```

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 PORC_SUBIDA_SENIOR_Y_SAL_BAJO      = 1.04,  
            PORC_SUBIDA_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){  
        porcentaje_actualizacion = PORC_SUBIDA_SENIOR_Y_SAL_BAJO;  
  
        if (es_familia_numerosa)  
            porcentaje_actualizacion = porcentaje_actualizacion  
                *  
                PORC_SUBIDA_FAMILIA_NUMEROSA;  
    }  
    else  
        porcentaje_actualizacion = PORC_SUBIDA_NO_SENIOR_Y_SAL_BAJO;  
}  
else  
    porcentaje_actualizacion = PORC_BAJADA_SIN_EXPERIENCIA;  
....
```

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



II.1.7.4. Las expresiones lógicas y el principio de una única vez

Según el principio de una única vez (página 115) 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 157. 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 cambiamos el criterio a que sea mayor estricto, debemos modificar el código de dos líneas. Para resolverlo, introducimos una variable lógica:

```
double nota_escrito;
bool escrito_superado;
.....
escrito_superado = nota_escrito >= 4.5; 
```

```
if (escrito_superado){
    nota_escrito = nota_escrito + 0.5;

    if (nota_escrito > 10)
        nota_escrito = 10;
}

.....
if (escrito_superado)
    cout << "Examen escrito superado con la nota: " << nota_escrito;
.....
```

El uso de variables intermedias para determinar criterios nos ayuda a no repetir código y facilita la legibilidad de éste. Se les asigna un valor en un bloque y se observa su contenido en otro.

II.1.7.5. Separación de entradas/salidas y cómputos

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 cómputos ya que, de esta forma:

- ▷ **Se separan responsabilidades.**

Cada módulo puede actualizarse de forma independiente.

- ▷ **Se favorece la reutilización entre plataformas (Linux, Windows, etc).**

Las E/S en Windows son distintas que en modo consola, por lo que los módulos serán distintos. Pero el módulo de cómputos será el mismo y podrá reutilizarse.

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

La siguiente norma es un caso particular de la indicada en la página 195

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

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 194. 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 variables intermedias.

```
int main(){
    const int MAYORIA_EDAD = 18,
              UMBRAL_ALTURA_JOVENES = 175,
              UMBRAL_ALTURA_ADULTOS = 190;
    int edad, altura;
    bool es_alto, es_mayor_edad, umbral_altura;

    // Entrada de datos:

    cout << "Introduzca los valores de edad y altura: ";
    cin >> edad;
    cin >> altura;

    // Cómputos:

    es_mayor_edad = edad >= MAYORIA_EDAD;

    if (es_mayor_edad)
        umbral_altura = UMBRAL_ALTURA_ADULTOS;
    else
        umbral_altura = UMBRAL_ALTURA_JOVENES;

    es_alto = altura >= umbral_altura;
```



```
// Salida de resultados:  
  
cout << "\n\n";  
  
if (es_mayor_edad)  
    cout << "Es mayor de edad";  
else  
    cout << "Es menor de edad";  
  
if (es_alto)  
    cout << "Es alto/a";  
else  
    cout << "No es alto/a";  
}
```

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

II.1.7.6. 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. Son tokens formados siguiendo las mismas reglas que las usadas para los nombres de los datos. Primero se define el **tipo** (lo haremos antes del `main`) y luego la variable de dicho tipo.

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

enum class CalificacionECTS
    {A, B, C, D, E, F, G}; // No van entre comillas!
enum class PuestoPodium
    {primero, segundo, tercero};
enum class 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.

Ejercicio. Modifique el código del programa que calcula las raíces de una ecuación de segundo grado (página 158) 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;

    // Cómputos:

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



```
if (radicando == 0){
    raiz1 = -b / denominador;
    tipo_ecuacion = TipoEcuacion::una_raiz_doble;
}
else{
    if (radicando > 0){
        radical = sqrt(radicando);
        raiz1 = (-b + radical) / denominador;
        raiz2 = (-b - radical) / denominador;
        tipo_ecuacion = TipoEcuacion::dos_raices_reales;
    }
    else
        tipo_ecuacion = TipoEcuacion::ninguna_raiz_real;
}
else{
    if (b != 0){
        raiz1 = -c / b;
        tipo_ecuacion = TipoEcuacion::recta_con_una_raiz;
    }
    else
        tipo_ecuacion = TipoEcuacion::no_es_ecuacion;
}

// Salida de Resultados:

cout << "\n\n";

switch (tipo_ecuacion){
    case TipoEcuacion::una_raiz_doble:
        cout << "Sólo hay una raíz doble: " << raiz1;
        break;
    case TipoEcuacion::dos_raices_reales:
```

```
    cout << "Las raíces son: " << raiz1 << " y " << raiz2;
    break;
case TipoEcuacion::ninguna_raiz_real:
    cout << "No hay raíces reales.";
    break;
case TipoEcuacion::recta_con_una_raiz:
    cout << "Es una recta. La única raíz es: " << raiz1;
    break;
case TipoEcuacion::no_es_ecuacion:
    cout << "No es una ecuación.";
    break;
}
}
```

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

Ampliación:

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

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

    salario_neto = 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.

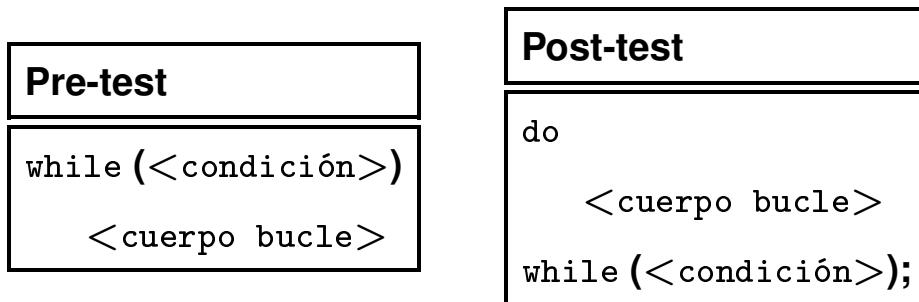
II.2. Estructuras repetitivas

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)**

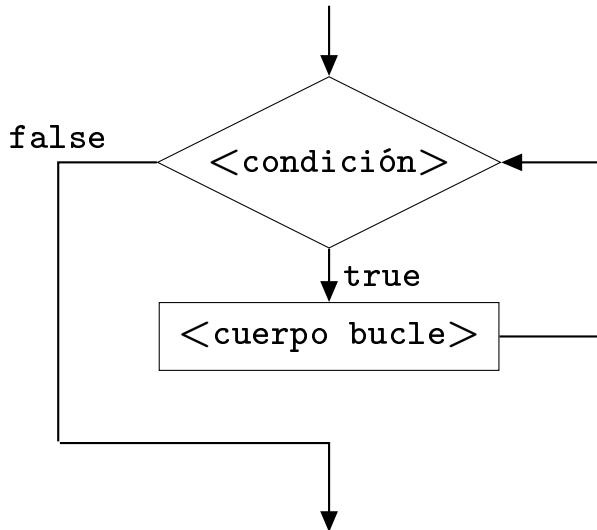
II.2.1. Bucles controlados por condición: pre-test y post-test

II.2.1.1. Formato

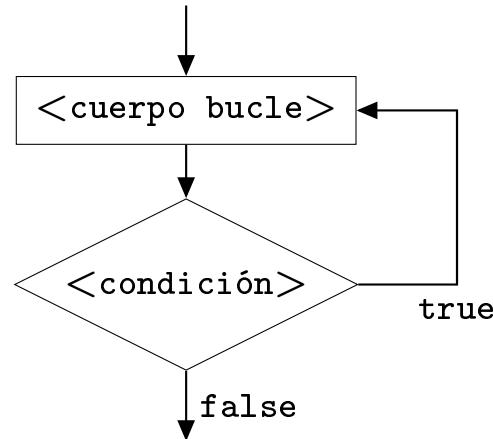


Funcionamiento: En ambos, se va ejecutando el cuerpo del bucle mientras la condición sea verdad.

- ▷ 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.



(a) while pre test



(b) do-while post test

Cada vez que se ejecuta el cuerpo del bucle diremos que se ha producido una *iteración (iteration)*

Al igual que ocurre en la estructura condicional, si los bloques de instrucciones contienen más de una línea, debemos englobarlos entre llaves. En el caso 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.

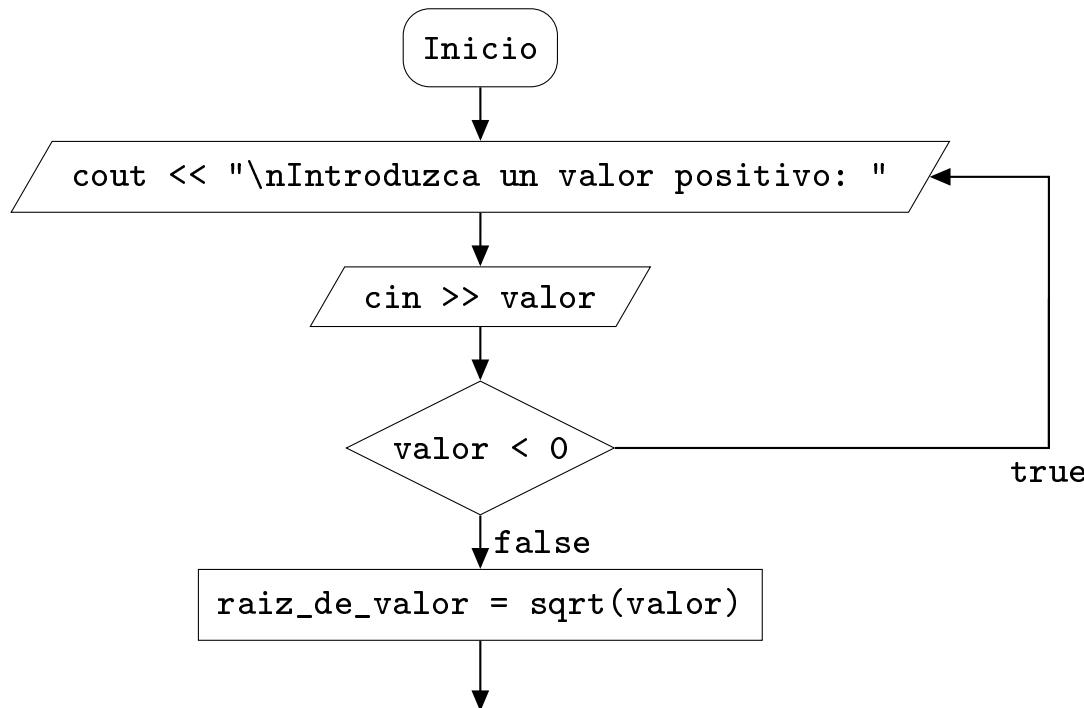
II.2.1.2. Algunos usos de los bucles

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 129)

Ejemplo. Escriba 20 líneas con 5 estrellas cada una.

```
.....
int num_lineas;

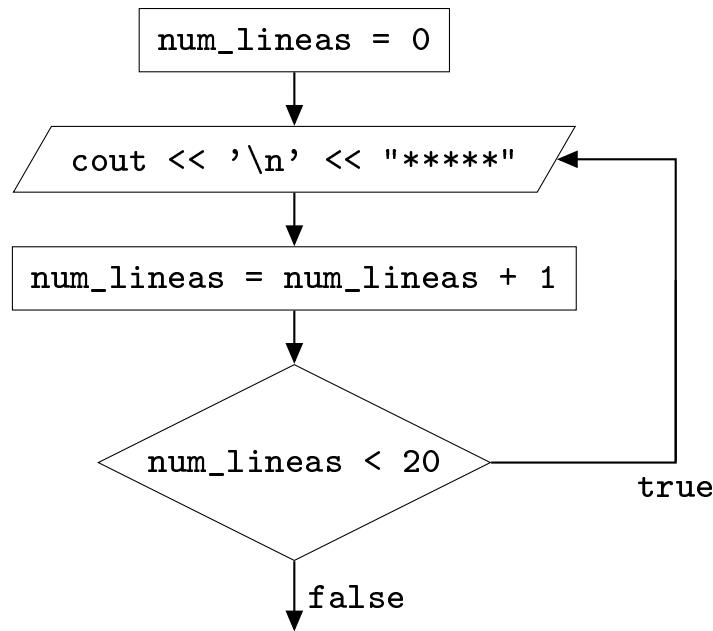
num_lineas = 1;

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

O bien:

```
num_lineas = 0;

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



O bien:

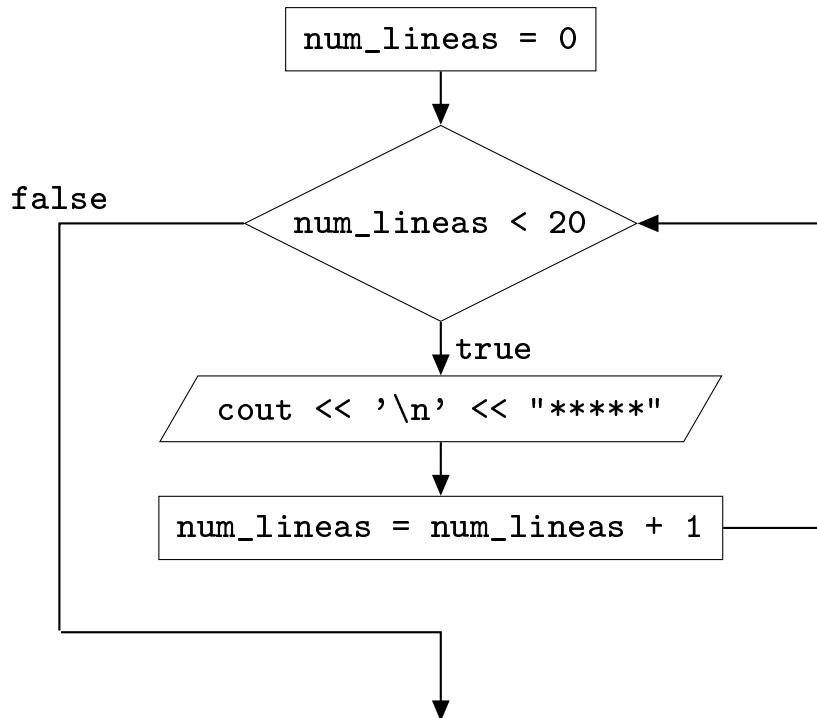
```
num_lineas = 1;

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

Preferible:

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

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



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

Nota:

Podríamos usar el operador de incremento:

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

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

Sí ⇒ pre-test

No ⇒ post-test

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

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

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

num_lineas = 0;

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

Problema: ¿Qué ocurre si `tope` 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.



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

```
num_digitos = 1;
```

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

Mejor si no modificamos la variable original:

```
n_dividido = abs(n); // valor absoluto  
num_digitos = 1;  
  
while (n_dividido > 9){  
    n_dividido = n_dividido/10;  
    num_digitos++;  
}
```

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

Ejemplo. Lea un entero `tope` positivo y escriba los pares \leq `tope`

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

par = 0;           // Primer candidato

while (par <= tope){
    par = par + 2;
    cout << par << " ";
}
```

Al final, escribe uno más. Cambiamos el orden de las instrucciones:

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

par = 0;           // Primer candidato

while (par <= tope){ // ¿Es bueno?
    cout << par;      // Si => Imprimelo
    par = par + 2;     // Calcular nuevo candidato
}                      // No => Salir
```

Si no queremos que salga 0, cambiaríamos la inicialización:

```
par = 2;           // Primer candidato
```

En el diseño de los bucles siempre hay que comprobar el correcto funcionamiento en los casos extremos (primera y última iteración)

II.2.1.3. Bucles con lectura de datos

En muchas ocasiones leeremos datos desde un dispositivo y tendremos que controlar una condición de parada. Habrá que controlar especialmente el primer y último valor leído.

Ejemplo. 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:

- ▷ Usando variables lógicas.
- ▷ Técnica de *lectura anticipada* . Leemos el primer valor antes de entrar al bucle y comprobamos si hay que procesarlo (el primer valor podría ser ya el terminador)

Solución con una variable lógica:

```
bool seguir_leyendo;  
  
do{  
    cin >> numero;                      // Leemos candidato  
  
    seguir_leyendo = (numero != TERMINADOR);  
  
    if (seguir_leyendo)                  // Comprobamos si es el terminador  
        suma = suma + numero;           // Lo procesamos  
}while (seguir_leyendo);
```



La expresión que controla la condición de parada (numero != TERMINADOR); sólo aparece en un único sitio, por lo que si cambiase el criterio de parada, sólo habría que cambiar el código en dicho sitio.

Es verdad que repetimos la observación de la variable seguir_leyendo (en el if y en el while) pero no repetimos la evaluación de la expresión anterior.

Solución con lectura anticipada:

```

suma = 0;
cin >> numero;           // Lectura anticipada del
                          // primer candidato
while (numero != TERMINADOR) { // Comprobamos si es el terminador
    suma = suma + numero;     // Lo procesamos
    cin >> numero;           // Leemos siguiente candidato
}

cout << "\nLa suma es " << suma;

```



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

A tener en cuenta:

- ▷ La primera vez que entra al bucle, la instrucción

while (numero != TERMINADOR)

hace las veces de un condicional. De esta forma, controlamos si hay que procesar o no el primer valor.

- ▷ Si el primer valor es el terminador, el algoritmo funciona correctamente.
- ▷ Hay cierto código repetido `cin >> numero;`, pero es aceptable. Mucho peor es repetir la expresión `numero != TERMINADOR` ya que es mucho más fácil que pueda cambiar en el futuro (porque cambie el criterio de parada)
- ▷ Dentro de la misma estructura repetitiva estamos mezclando las entradas (`cin >> numero;`) de datos con los cálculos (`suma = suma + numero`), violando lo visto en la página 199. 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, o bien usaremos la técnica de lectura anticipada o bien introduciremos variables lógicas que controlen la condición de terminación de lectura.

Ejemplo. Lea enteros hasta llegar al cero. Imprima el número de pares e impares leídos.

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

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

    while (valor != 0){
        if (valor % 2 == 0)
            num_pares++;
        else
            num_impares++;

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

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

Ejemplo. Retome el ejemplo de la subida salarial de la página 196. 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/~carlos/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?

```
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?

*Evite siempre el uso de **números mágicos (magic numbers)**, es decir, literales numéricos cuya función en el código no queda clara.*

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.

Intentamos sacar esa comprobación fuera del bucle.

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/~carlos/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;
    }
```

En cualquier caso, 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. ¿Cuál es el operador que cumple lo siguiente?:

false	Operador	<lo que sea>	=	false
true	Operador	true	=	true

Es el operador `&&`

Mejor aún:

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

Ejemplo. 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/~carlos/FP/II_maximo_comun_divisor.cpp

Ambas soluciones son equivalentes. Formalmente:

$a \% \text{ divisor} != 0 || b \% \text{ divisor} != 0$
equivale a
 $! (a \% \text{ divisor} == 0 \&\& b \% \text{ divisor} == 0)$

Diseñe las condiciones compuestas de forma que sean fáciles de leer

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 encontramos 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;  
bool es_primo;  
  
cout << "Introduzca un numero natural: ";  
cin >> valor;  
  
es_primo = true;  
divisor = 2  
  
while (divisor < valor){  
    if (valor % divisor == 0)  
        es_primo = false;  
  
    divisor++;  
}  
  
if (es_primo)  
    cout << valor << " es primo\n";  
else{  
    cout << valor << " no es primo\n";  
    cout << "\nSu primer divisor es: " << divisor;  
}
```



Funciona pero es ineficiente. Nos debemos salir del bucle en cuanto se pamos que no es primo. Usamos la variable es_primo.

```
int main(){
    int valor, divisor;
    bool es_primo;

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

    es_primo = true;
    divisor = 2;

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

    if (es_primo)
        cout << valor << " es primo";
    else{
        cout << valor << " no es primo";
        cout << "\nSu primer divisor es: " << divisor;
    }
}
```

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

Los algoritmos que realizan una búsqueda, deben salir de ésta en cuanto se haya encontrado el valor. Normalmente, usaremos una variable lógica para controlarlo.

Nota:

Al usar else garantizamos que al salir del bucle, la variable divisor contiene el primer divisor de valor

Ampliación:

Incluso podríamos quedarnos en sqrt(valor), ya que si valor no es primo, tiene al menos un divisor menor que sqrt(valor). En cualquier caso, sqrt es una operación costosa y habría que evaluar la posible ventaja en su uso.

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

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

II.2.2. Programando como profesionales

II.2.2.1. Evaluación de expresiones dentro y fuera del bucle

En ocasiones, una vez terminado un bucle, debemos comprobar cuál fue la condición que hizo que éste terminase. Veamos cómo hacerlo correctamente.

Ejemplo. Desde un sensor se toman datos de la frecuencia cardíaca de una persona. Emite una alarma cuando se encuentren fuera del rango [45, 120] indicando si es baja o alta.

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

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

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

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

        cout << latidos;
    }
```

Se repite código ya que latidos < MIN_LATIDOS equivale a



!(MIN_LATIDOS <= latidos) Para resolverlo, introducimos variables lógicas intermedias:

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

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

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

    cout << latidos;
}
```

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

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

Si una vez que termina un bucle controlado por varias condiciones, necesitamos saber cuál de ellas hizo que termine 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 cuatro. En cualquier caso, el cuarto valor (en general el último) hay que procesarlo fuera del bucle.

```
suma = 0;  
cin >> nota;  
total_introducidos = 1;  
  
while (nota >= 0 && total_introducidos < 4){  
    suma = suma + nota;  
    cin >> nota;  
    total_introducidos++;  
}  
  
suma = suma + nota;  
media = suma/total_introducidos;
```



Problema: Si el valor es negativo lo suma.

```
suma = 0;  
cin >> nota;  
total_introducidos = 1;  
  
while (nota >= 0 && total_introducidos < 4){  
    suma = suma + nota;  
    cin >> nota;  
    total_introducidos++;  
}  
  
if (nota > 0)  
    suma = suma + nota;  
  
media = suma/total_introducidos;
```



247

Además, no debemos aumentar total_introducidos si es un negativo:

```
if (nota > 0)  
    suma = suma + nota;  
else  
    total_introducidos--;  
  
media = suma/total_introducidos;
```



247

Podemos observar la complejidad (por no decir chapucería) innecesaria que ha alcanzado el programa.

Replanteamos desde el inicio la solución y usamos variables lógicas:

```
int main(){

    const int TOPE_NOTAS = 4;
    int nota, suma, total_introducidos;
    double media;
    bool tope_alcanzado, es_correcto;

    cout << "Introduzca un máximo de " << TOPE_NOTAS
        << " notas, o cualquier negativo para finalizar.\n ";

    suma = 0;
    total_introducidos = 0;
    es_correcto = true;
    tope_alcanzado = false;

    do{
        cin >> nota;

        if (nota < 0)
            es_correcto = false;
        else{
            suma = suma + nota;
            total_introducidos++;

            if (total_introducidos == TOPE_NOTAS)
                tope_alcanzado = true;
        }
    }while (es_correcto && !tope_alcanzado);

    media = suma/(1.0 * total_introducidos); // Si total_introducidos es 0
                                                // media = infinito

    if (total_introducidos == 0)
        cout << "\nNo se introdujo ninguna nota";
    else
```



```
cout << "\nMedia aritmética = " << media;  
}
```

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

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 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;

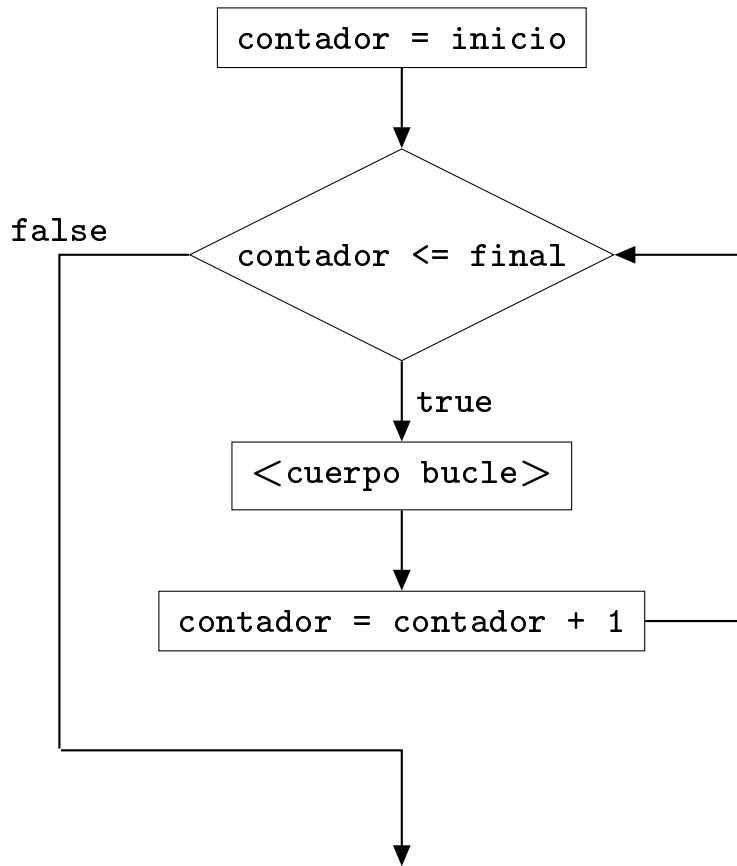
    inicio = 1;
    final = 5;
    suma = 0;
    contador = inicio;

    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:

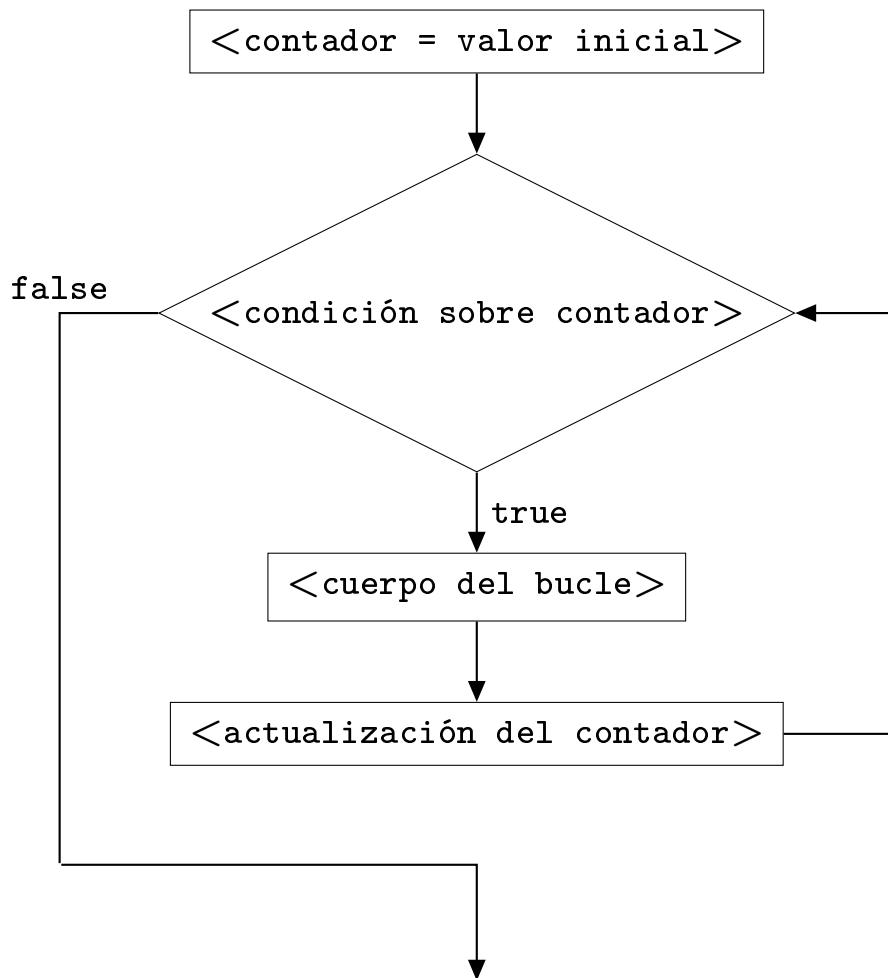


Vamos a implementar el mismo diagrama, pero con otra sintaxis (usando el bucle `for`).

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>
```



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

Ejemplo. Imprima 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 << " ";  
}
```

Este problema también se podría haber resuelto como sigue:

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

Ejemplo. 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, inicio ≤ final y contador <= final`

final - inicio + 1

▷ Si `incremento = 1, inicio ≤ final y contador < final`

final - inicio

```
for (i = 0; i < 10; i++)    --> 10 - 0 = 10  
    cout << "*";
```

```
for (i = 12; i > 2; i--)    --> 12 - 2 = 10  
    cout << "*";
```

```
for (i = 10; i >= 1; i--)    --> 10 - 1 + 1 = 10  
    cout << "*";
```

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

Nota:

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

Ampliación:

Número de iteraciones con incrementos cualesquiera.

Si es del tipo contador `<= final`, tenemos que contar cuántos intervalos de longitud igual a incremento hay entre los valores inicio y final. El número de iteraciones será uno más.

En el caso de que contador `< final`, habrá que contar el número de intervalos entre inicio y final - 1.

El número de intervalos se calcula a través de la división entera.

En resumen, considerando incrementos positivos:

- ▷ Si el bucle es del tipo contador `<= final` el número de iteraciones es `(final - inicio) / 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 `(final - 1 - inicio) / incremento + 1` siempre y cuando sea `inicio < final`. En otro caso, hay 0 iteraciones.
 - ▷ De forma análoga se realizan los cálculos con incrementos negativos (en cuyo caso, el valor inicial ha de ser mayor o igual que el final).
-

Ejercicio. ¿Qué salida producen los siguientes trozos de código?

```
int i, suma_total;  
suma_total = 0;  
  
for (i = 1 ; i <= 10; i++)  
    suma_total = suma_total + 3;
```

```
suma_total = 0;  
  
for (i = 5 ; i <= 36 ; i++)  
    suma_total++;
```

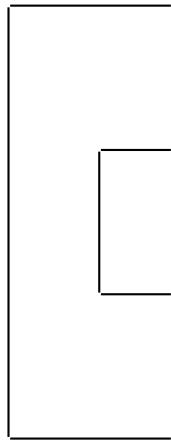
```
suma_total = 0;  
  
for (i = 5 ; i <= 36 ; i = i+1)  
    suma_total++;
```

```
suma_total = 0;  
i = 5;  
  
while (i <= 36){  
    suma_total++;  
    i = i+1;  
}
```

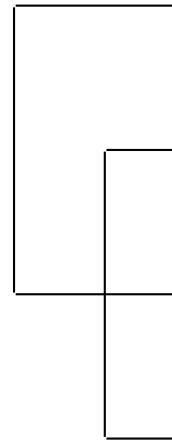
II.2.4. Anidamiento de bucles

Dos bucles se encuentran anidados, cuando uno de ellos está en el bloque de sentencias del otro.

En principio no existe límite de anidamiento, y la única restricción que se debe satisfacer es que deben estar completamente inscritos unos dentro de otros.



(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";

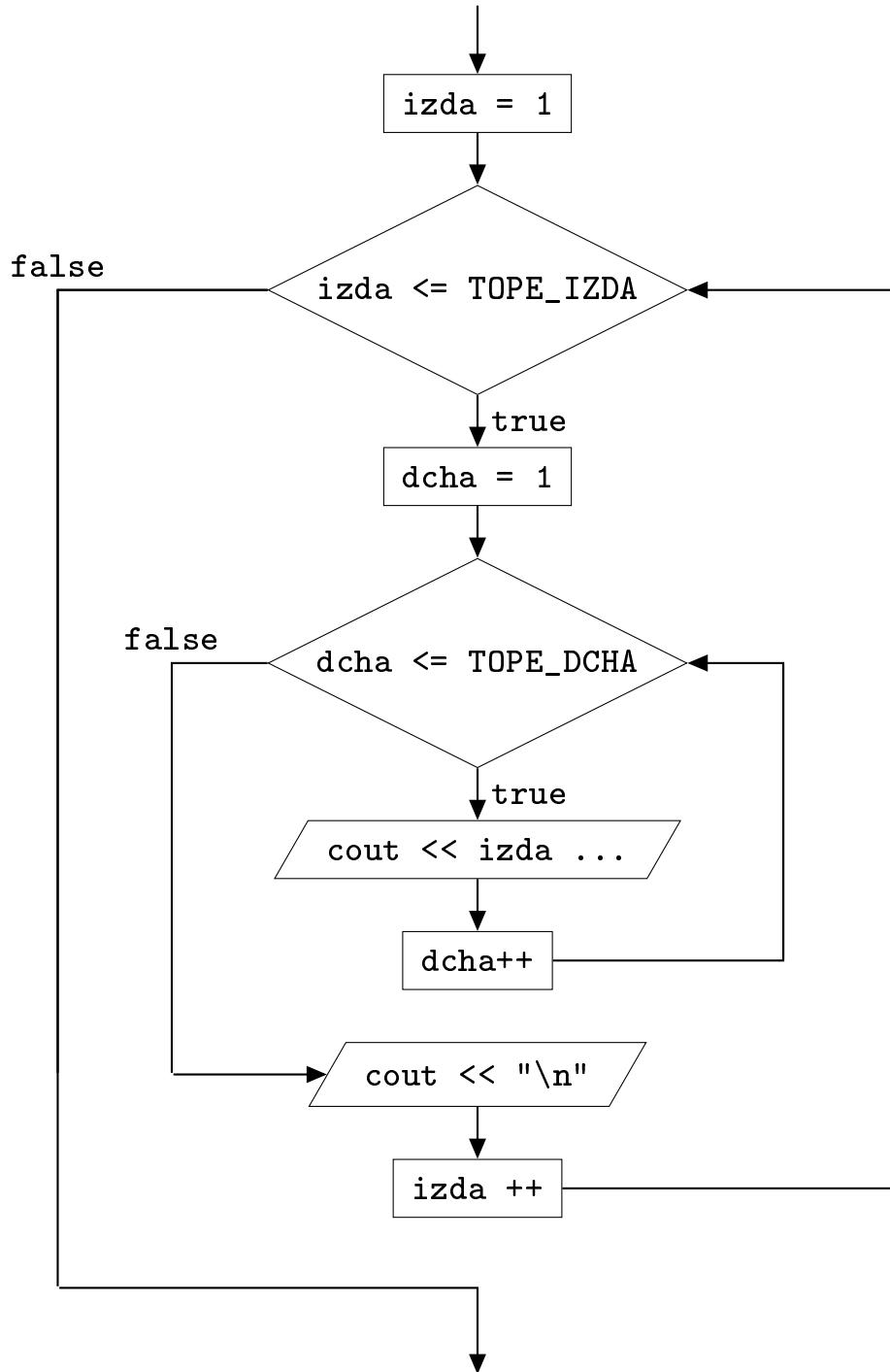
    for (izda = 1 ; izda <= TOPE_IZDA ; izda++) {
        for (dcha = 1 ; dcha <= TOPE_DCHA ; dcha++)
            cout << izda << "*" << dcha << "=" << izda * dcha << "   ";
        cout << "\n";
    }
}
```

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

	dcha=1	dcha=2	dcha=3
izda=1	1*1 = 1	1*2 = 2	1*3 = 3
izda=2	2*1 = 2	2*2 = 4	2*3 = 6
izda=3	3*1 = 3	3*2 = 6	3*3 = 9

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
```

`suma = 5 * (1 + 2 + 3 + 4 + 5)`. En general:

$$\text{suma} = n \sum_{i=1}^{i=n} i = n \frac{n^2 + n}{2} = \frac{n^3 + n^2}{2}$$

Si n es 5, `suma` se quedará con 75

Ejemplo. ¿Qué salida produce el siguiente código?

```

iteraciones = 0;
suma = 0;

for (i = 1 ; i <= n; i++){
    for (j = i ; j <= n; j++){
        suma = suma + j;
        iteraciones++;
    }
}

```

Número de iteraciones:

$$n + (n - 1) + (n - 2) + \dots + 1 = \sum_{i=1}^{i=n} i = \frac{n^2 + n}{2} < n^2$$

Valor de la variable suma. Supongamos n = 5. Valores que va tomando j:

```

1 + 2 + 3 + 4 + 5 +
2 + 3 + 4 + 5 +
3 + 4 + 5 +
4 + 5 +
5

```

$$\text{suma} = 5 * 5 + 4 * 4 + 3 * 3 + 2 * 2 + 1 * 1 = \sum_{i=1}^{i=n} i^2 = \frac{1}{6}n(n+1)(2n+1)$$

Si n es 5, suma se quedará con 55

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

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

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

/*
    Recorremos todos los números menores que entero
        Comprobamos si dicho número es primo
*/
for (possible_primo = entero - 1 ; posible_primo > 1 ; posible_primo--){
    es_primo = true;
    divisor = 2;

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

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

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

Ejemplo. El Teorema fundamental de la Aritmética (Euclides 300 A.C/Gauss 1800) nos dice que podemos expresar cualquier entero como producto de factores primos.

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: $n \geq primo$ 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 241. 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/~carlos/FP/II_descomposicion_en_primos.cpp

II.3. Particularidades de C++

C++ es un lenguaje muy versátil. A veces, demasiado . . .

II.3.1. Expresiones y sentencias son similares

II.3.1.1. El tipo `bool` como un tipo entero

En C++, el tipo lógico es compatible con un tipo entero. Cualquier expresión entera que devuelva el cero, se interpretará como `false`. Si devuelve cualquier valor distinto de cero, se interpretará como `true`.

```
bool var_logica;

var_logica = false;
var_logica = (4 > 5); // Correcto: resultado false
var_logica = 0;       // Correcto: resultado 0 (false)

var_logica = (4 < 5); // Correcto: resultado true
var_logica = true;
var_logica = 2;        // Correcto: resultado 2 (true)
```

Nota. Normalmente, al ejecutar `cout << false`, se imprime en pantalla un cero, mientras que `cout << true` imprime un uno.

La dualidad entre los tipos enteros y lógicos nos puede dar quebraderos de cabeza en los condicionales

```
int dato = 4;

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



El operador ! tiene más precedencia que <. Por lo tanto, la evaluación es como sigue:

$$\begin{aligned} ! \text{dato} < 5 &\Leftrightarrow (\text{!dato}) < 5 \Leftrightarrow (\text{4 equivale a true}) (\text{!true}) < 5 \Leftrightarrow \\ &\Leftrightarrow \text{false} < 5 \Leftrightarrow 0 < 5 \Leftrightarrow \text{true} \end{aligned}$$

¡Imprime 4 es mayor o igual que 5!

Solución:

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

o mejor, simplificando la expresión siguiendo el consejo de la página 176

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

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

```
bool es_menor;

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

II.3.1.2. El operador de asignación en expresiones

El operador de asignación = se usa en sentencias del tipo:

```
valor = 7;
```

Pero además, devuelve un valor: el resultado de la asignación. Así pues, valor = 7 **es una expresión** que devuelve 7

```
un_valor = otro_valor = valor = 7;
```

Esto producirá fuertes dolores de cabeza cuando por error usemos una expresión de asignación en un condicional:

```
valor = 5;
```

```
if (valor = 7)
    <acciones if> // Siempre se ejecuta este bloque!
else
    <acciones else>

// Además, valor se queda con 7
```



valor = 7 **devuelve 7. Al ser distinto de cero, es true. Por tanto, se ejecuta el bloque if (y además valor se ha modificado con 7)**

Otro ejemplo:

```
a = 7;
```

```
if (a = 0)
    cout << "\nRaíz= " << -c/b; // Nunca se ejecuta!
else{
    r1 = -b + sqrt(b*b - 4*a*c) / (2*a) ; // Error lógico
```

II.3.1.3. El operador de igualdad en sentencias

C++ permite que una expresión constituya una sentencia 😕

Esta particularidad no da beneficios salvo en casos muy específicos y sin embargo nos puede dar quebraderos de cabeza. Así pues, el siguiente código compila perfectamente:

```
int entero;  
4 + 3;
```

C++ evalúa la expresión **entera** `4 + 3;`, devuelve 7 y no hace nada con él, prosiguiendo la ejecución del programa.

Otro ejemplo:

```
int entero;  
entero == 7;
```

C++ evalúa la expresión **lógica** `entero == 7;`, devuelve true y no hace nada con él, prosiguiendo la ejecución del programa.

II.3.1.4. El operador de incremento en expresiones

El operador `++` (y `--`) puede usarse dentro de una expresión.

Si se usa en forma postfija, primero se evalúa la expresión y luego se incrementa la variable.

Si se usa en forma prefija, primero se incrementa la variable y luego se evalúa la expresión.

Ejemplo. El siguiente condicional expresa una condición del tipo *Comprueba si el siguiente es igual a 10*

```
variable = 9;  
if (variable + 1 == 10)  
    cout << variable;           // Imprime 9  
cout << " " << variable;     // Imprime 9
```

El siguiente condicional expresa una condición del tipo *Comprueba si el actual es igual a 10 y luego increméntalo*

```
variable = 9;  
if (variable++ == 10)  
    cout << variable;         // No entra  
cout << " " << variable;     // Imprime 10
```

El siguiente condicional expresa una condición del tipo *Incrementa el actual y comprueba si es igual a 10*

```
variable = 9;  
if (++variable == 10)  
    cout << variable;         // Imprime 10  
cout << " " << variable;     // Imprime 10
```

Consejo: *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;
```

II.3.2.2. Bucles for con sentencias de incremento incorrectas

Lo siguiente es un error lógico:

```
for (par = -10; par <= 10; par + 2) // en vez de par = par + 2  
    num_pares++;
```

equivale a:

```
par = -10;  
  
while (par <= 10){  
    num_pares++;  
    par + 2;  
}
```

Compila correctamente pero la sentencia par + 2; no incrementa par (recuerde lo visto en la página 272) Resultado: bucle infinito.

II.3.2.3. Modificación del contador

Únicamente mirando la cabecera de un bucle `for` sabemos cuantas iteraciones se van a producir (recordar lo visto en la página 257). Por eso, en los casos en los que sepamos de antemano cuántas iteraciones necesitamos, usaremos un bucle `for`. En otro caso, usaremos un bucle `while` ó do `while`.

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

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

IMPORTANT

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

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

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

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

Solución:

```
suma = 0;  
tope = valor/2;  
  
for (divisor = 2; divisor <= tope ; divisor++) {  
    if (valor % divisor == 0)  
        suma = suma + 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 pero es muy oscuro. 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;
```



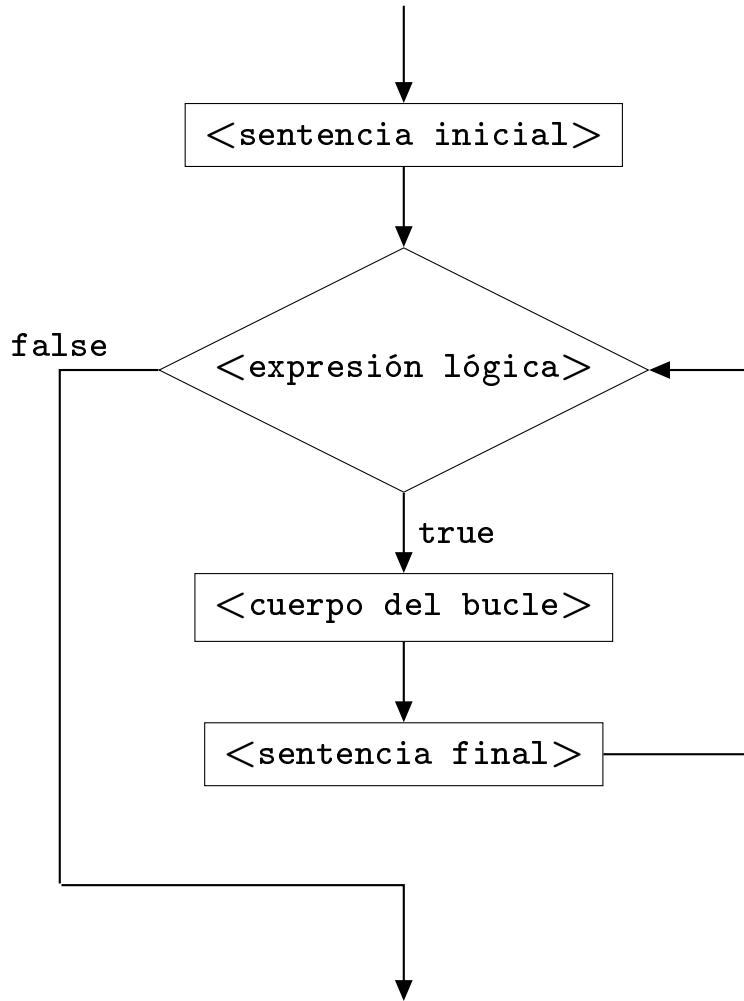
Nunca nos saldremos de un bucle for asignándole un valor extremo a la variable contadora.

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 arriba para comprobar el valor de la expresión lógica.



Por tanto, la condición impuesta en el ciclo no tiene por qué ser de la forma contador < final, sino que puede ser cualquier tipo de condición. Veamos en qué situaciones es útil esta flexibilidad.

Ejemplo. 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 278 como ésta son correctas.

Ejemplo. Compruebe si un número es primo. Lo resolvimos en la página 241:

```
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: *Límite 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 muy 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 anteriores, excepto la sentencia `break` con el propósito aquí descrito dentro de la sentencia `switch`.

IMPORTANT

Tema III

Funciones

Objetivos:

- ▷ Introducir el principio de ocultación de información.
- ▷ Introducir el concepto de función y parámetro para no tener que repetir código.
- ▷ Conocer el mecanismo por el que se llama o ejecuta una función, y cómo se retorna al punto en el que se hizo la llamada.
- ▷ Profundizar en el ámbito de los datos.
- ▷ Introducir el concepto de precondición.

Autor: Juan Carlos Cubero.

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

III.1. Fundamentos

III.1.1. Las funciones realizan una tarea

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

- ▷ Identificar las acciones repetidas.
- ▷ Identificar los valores que pueden variar entre las diferentes repeticiones (esto es, los *parámetros*).

Finalmente se define una función que encapsula dichas acciones y permita su ejecución con diferentes parámetros.

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

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

```
int main(){  
    double lado1, lado2, hipotenusa, radicando;  
  
    <Asignación de valores a los lados>  
  
    radicando = lado1*lado1 + lado2*lado2;  
    hipotenusa = sqrt(radicando);
```

Si queremos calcular la hipotenusa de dos triángulos rectángulos:

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

    <Asignación de valores a los lados>

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

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

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

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

    <Asignación de valores a los lados>

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

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

III.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.
- ▷ Diremos que

`<tipo> <nombre-función> (<parám.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. Son los datos genéricos. Cuando se está ejecutando la función contendrán los valores concretos con los que se invoca la ejecución de la función.

- ▷ El cuerpo de la función debe contener¹:

`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 de la expresión es el valor que devuelve la función cuando termina su ejecución

¹excepto si se trata de una función `void` (no devuelve nada)

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

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:
`Cuadrado(valor)` es una expresión.
- ▷ En C++ no se pueden definir funciones dentro de otras. Todas están al mismo nivel.
- ▷ Como estilo de codificación, escribiremos la primera letra de las funciones en mayúscula.

III.1.3. Parámetros formales y reales

- ▷ 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.
- ▷ Los **parámetros reales (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.

Nota:

En muchos manuales puede encontrar el término **parámetro *actual*** en lugar de parámetro **real**. Se trata de una traducción algo forzada.

```
double Cuadrado(double entrada){      // entrada: parámetro formal
    return entrada*entrada;
}
int main(){
    double resultado, valor;

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

► Flujo de control

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

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

$\langle \text{parámetro formal} \rangle = \langle \text{parámetro actual} \rangle$

En el ejemplo, `entrada = 4`

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

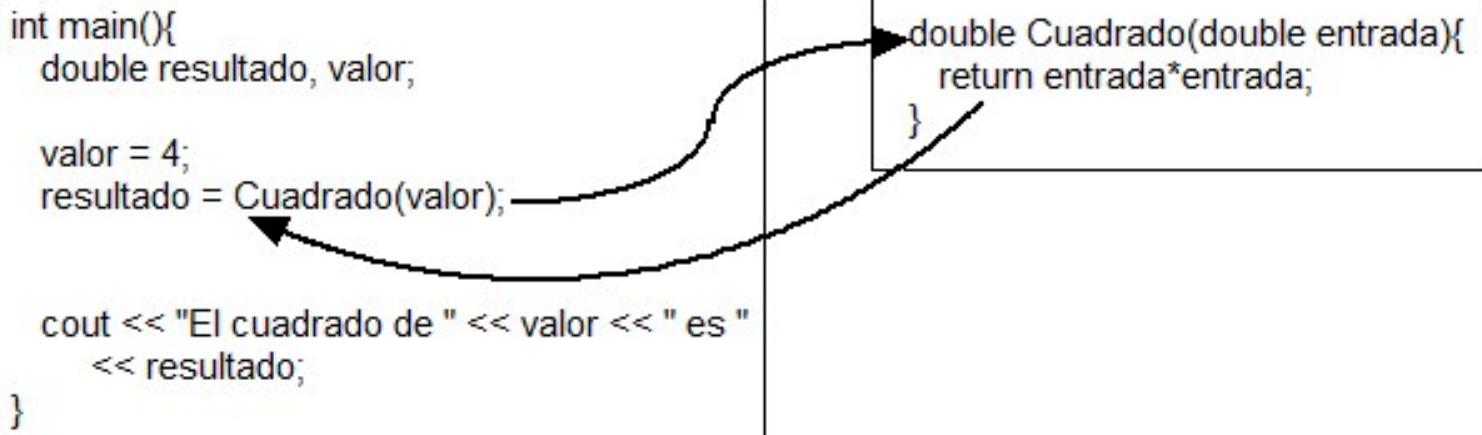
- ▷ Empiezan a ejecutarse las sentencias de la función y, cuando se llega a alguna sentencia `return <expresión>`, termina la ejecución de la función y devuelve el resultado de evaluar `<expresión>` al lugar donde se realizó la invocación.
- ▷ A continuación, el flujo de control prosigue por donde se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

double Cuadrado(double entrada){

entrada = valor

```
    return entrada*entrada;
}
int main(){
    double resultado, valor;

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



► Correspondencia entre parámetros actuales y formales

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

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

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

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

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

```
double Resta(double valor_1, double valor_2){  
    return valor_1 - valor2;  
}  
  
int main(){  
    double un_valor = 5.0, otro_valor = 4.0;  
    double resta;  
  
    resta = Resta(un_valor, otro_valor); // 1.0  
    resta = Resta(otro_valor, un_valor); // -1.0
```

- ▷ El parámetro actual puede ser una expresión.

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

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

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

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

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

```
int DivisonEntera(int numerador, int denominador){  
    return numerador / denominador;  
}  
  
int main(){  
    ... DivisonEntera(400000000000, 10); // Desbordamiento
```



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

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

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

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

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

    <Asignación de valores a los lados>

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

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

Flujo de control:

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

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

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

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

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

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

Ejercicio. Defina la función Hipotenusa

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

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

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

    <Asignación de valores a los lados>

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

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

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

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

Ejercicio. 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/~carlos/FP/III_par.cpp

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

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



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

En resumen:

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

▷ ***Menos propenso a errores***

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

▷ ***Más fácil de mantener***

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

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

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

IMPORTANT

III.1.4. Ámbito de un dato. Datos locales

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

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

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

```
<tipo> <nombre-función> (<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 variable i y $fact$. Éstas no se conocen fuera de la función.

```
#include <iostream>  
using namespace std;  
  
long long Factorial (int n){  
    int i;  
    long long fact = 1;  
  
    for (i = 2; i <= n; i++)  
        fact = fact * i;  
  
    return fact;  
}
```

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

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

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

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

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

```
long long Factorial (int n){  
    int i;  
    long long fact = 1;  
  
    for (i = 2; i <= valor; i++) // Error de compilación   
        fact = fact * i;  
  
    return fact;  
}  
  
int main(){  
    int valor:  
    int resultado;  
  
    cout << i; // Error de compilación   
    n = 5; // Error de compilación   
  
    cout << "\nIntroduzca valor";  
    cin >> valor;  
  
    resultado = Factorial(valor);  
    cout << "\nFactorial de " << valor << " = " << resultado;  
}
```

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

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

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

    return fact;
}

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

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

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

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

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

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

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

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

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

Ejercicio. Compruebe si un número es primo (recuerde el algoritmo visto en la página 241)

```
bool EsPrimo(int valor){  
    bool es_primo;  
    int divisor;  
  
    es_primo = true;  
  
    for (divisor = 2 ; divisor < valor && es_primo ; divisor++)  
        if (valor % divisor == 0)  
            es_primo = false;  
  
    return es_primo;  
}  
  
int main(){  
    int un_numero;  
    bool es_primo;  
  
    cout << "Comprobar si un número es primo.\n\n";  
    cout << "Introduzca un entero: ";  
    cin >> un_numero;  
  
    es_primo = EsPrimo(un_numero);  
  
    if (es_primo)  
        cout << un_numero << " es primo";  
    else  
        cout << un_numero << " no es primo";  
}
```

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

Ejercicio. 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/~carlos/FP/III_mcd_funcion_1.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 una entrada sobre un tipo `string` y analizar la cadena.

III.1.5. La Pila

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

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

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

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

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

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

Construimos la función Combinatorio que llama a la función Factorial:

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

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

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

    return fact;
}

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

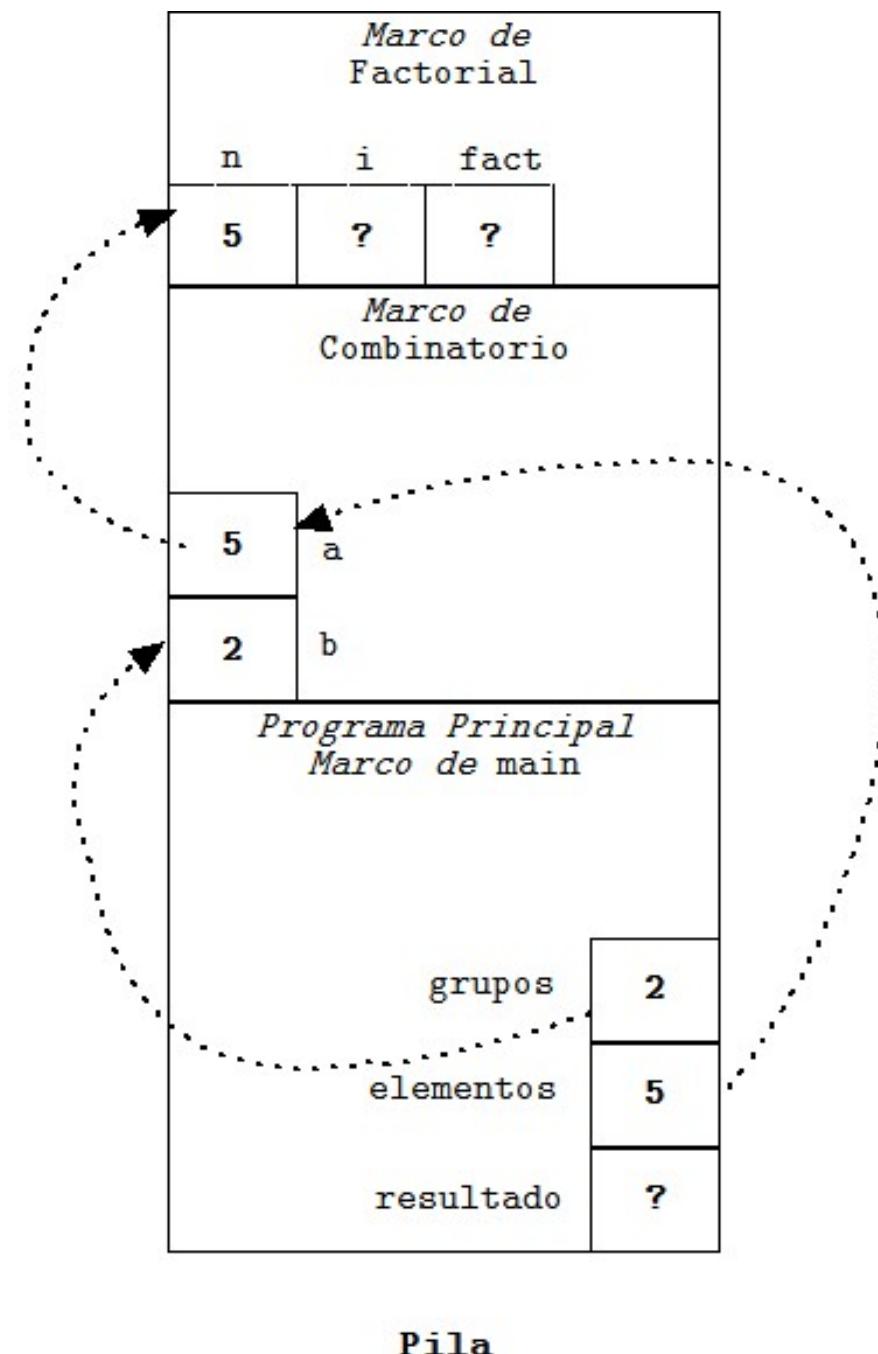
int main(){
```

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

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

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

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

Ampliación:



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

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

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

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

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

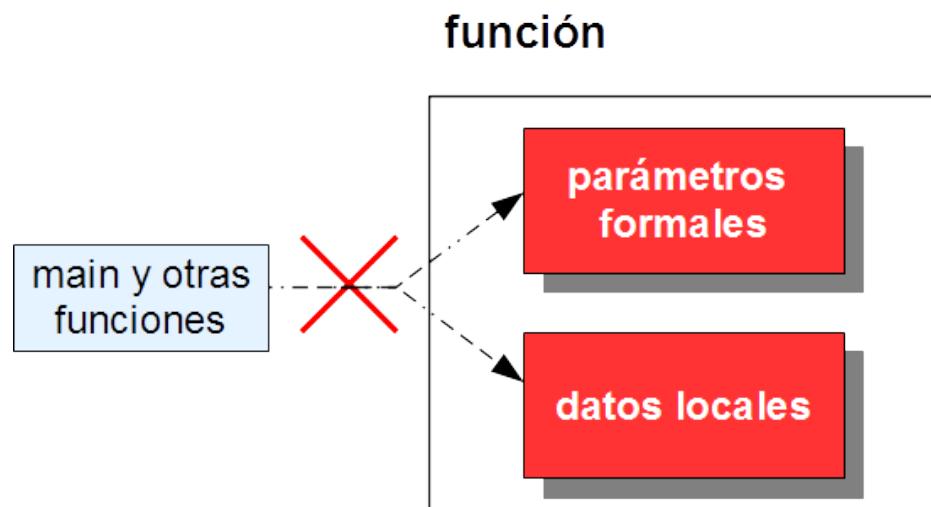
En C++, si la función `main` no incluye una sentencia `return` y termina de ejecutarse correctamente el programa, se devuelve 0 por defecto, por lo que podríamos suprimir `return 0;` (sólo en la función `main`)

III.2. El principio de ocultación de información

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

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

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



Al llamar a una función no nos preocupamos de saber cómo realiza su tarea. Esto nos permitirá, por ejemplo, cambiar la implementación de la función sin tener que modificar nada en la función que hace la llamada.

Ejemplo. Sobre el ejercicio que comprueba si un número es primo (página 311) podemos mejorar la implementación, parando al llegar a la raíz cuadrada del valor sin haber encontrado un divisor (ver página 242)

```
bool EsPrimo(int valor){  
    bool es_primo;  
    int divisor;  
    double tope;  
  
    es_primo = true;  
    tope = sqrt(valor);  
  
    for (divisor = 2 ; divisor <= tope && es_primo ; divisor++)  
        if (valor % divisor == 0)  
            es_primo = false;  
  
    return es_primo;  
}  
  
int main(){  
    .....  
    No cambia nada  
    .....  
}
```

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

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

Ejemplo. Retomamos el ejemplo del número combinatorio de la página 316. 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:

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

Cambiemos la implementación de la función como sigue:

```
long long Combinatorio(int a, int b){  
    int numerador, fact_b;  
  
    numerador = 1;  
    fact_b = 1;  
  
    for (int i = 1 ; i <= b ; i++){  
        fact_b = fact_b*i;  
        numerador = numerador * (a - i + 1);  
    }  
  
    return numerador / fact_b;  
}  
  
int main(){  
    No cambia nada  
}
```

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

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

Principio de Programación:

Ocultación de información (Information Hiding)

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



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

III.3. Funciones void

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

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

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

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

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

```
void <nombre-función> (<parámetros formales>) {  
    <constantes locales>  
    <variables locales>  
    <sentencias>  
}
```

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

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

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

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

Nota:

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

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

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

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

void Presentacion(int tope_lineas){
    int i;
    for (i = 1; i <= tope_lineas ; i++)
        cout << "\n*****";
    cout << "Programa básico de Trigonometría";
    for (i = 1; i <= tope_lineas ; i++)
        cout << "\n*****";
}
int main(){
    double lado1, lado2, hipotenusa;

    Presentacion(3);

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

    hipotenusa = Hipotenusa(lado1,lado2);

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

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

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

```
void ImprimeLineas (int num_lineas){  
    int i;  
    for (i = 1; i <= num_lineas ; i++)  
        cout << "\n*****";  
}  
  
void Presentacion(int tope_lineas){  
    ImprimeLineas (tope_lineas);  
    cout << "Programa básico de Trigonometría";  
    ImprimeLineas (tope_lineas);  
}
```

III.4. Ámbito de un dato (revisión)

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

► *Lo que ya sabemos:*

Datos locales (declarados dentro de una función)

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

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

Parámetros formales de una función

- ▷ Se consideran datos locales de la función.

► **Lo nuevo (ni bueno ni malo):**

Datos declarados dentro de un bloque

- ▷ El ámbito es el propio bloque. El ámbito de la variable termina con la llave } que cierra la estructura condicional, repetitiva, etc.
 - En un bucle controlado por condición, la variable pierde el valor antiguo en cada iteración:

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

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

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

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

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

Cuando necesitemos puntualmente variables intermedias para realizar nuestros cálculos, será útil 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.

Debemos tener cuidado con la declaración de variables con igual nombre que otra definida en un ámbito *superior*.

Prevalece la de ámbito más restringido → *prevalencia de nombre (name hiding)*

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



► **Lo nuevo (muy malo):**

Datos globales (global data)

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

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

Ejemplo. Supongamos un programa para la gestión de un aeropuerto. Tendrá dos funciones: GestiónMaletas 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.

IMPORTANT

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

III.5. Parametrización de funciones

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

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

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

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

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

Ejemplo. Retomemos el ejemplo ImprimeLineas de la página 327.

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

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

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

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

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

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

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

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

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

```
void ImprimeAsteriscos (int num_asteriscos){  
    for (int i = 1 ; i <= num_asteriscos ; i++)  
        cout << "*";  
}  
  
void ImprimeLineas (int num_lineas, int num_asteriscos){  
    for (int i = 1; i <= num_lineas ; i++){  
        cout << "\n";  
        ImprimeAsteriscos(num_asteriscos);  
    }  
}  
  
void Presentacion(int tope_lineas, int num_asteriscos){  
    ImprimeLineas (tope_lineas, num_asteriscos);  
    cout << "Programa básico de Trigonometría";  
    ImprimeLineas (tope_lineas, num_asteriscos);  
}
```

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

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

El programa principal quedaría así:

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

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

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

    hipotenusa = Hipotenusa(lado1, lado2);

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

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

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

Ejemplo. Retomemos el ejemplo de la altura del tema II. Si el criterio es el de la página 152, 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 << "Mayría 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 203:

```
bool EsMayorEdad (int edad){  
    const int MAYORIA_EDAD = 18;  
  
    return edad >= MAYORIA_EDAD;  
}  
  
bool EsAlta (int altura, int edad){  
    const int UMBRAL_ALTURA_JOVENES = 175,  
            UMBRAL_ALTURA_ADULTOS = 190;  
    int umbral_altura;  
  
    if (EsMayorEdad(edad))  
        umbral_altura = UMBRAL_ALTURA_ADULTOS;  
    else  
        umbral_altura = UMBRAL_ALTURA_JOVENES;  
  
    return altura >= umbral_altura;  
}  
  
int main(){  
    <No cambia nada>  
}
```

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

Nota:

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

III.6. Programando como profesionales

III.6.1. Cuestión de estilo

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

Ejemplo. ¿Qué ocurre en el siguiente código?

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

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

Ejemplo. 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.*



III.6.2. Diseño de la cabecera de una función

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

Ejemplo. Faltan parámetros:

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

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

    cin >> n;

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

    return fact;
}

int main(){
    int resultado;

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

IMPORTANT

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);
    .....
}
```



Ejemplo. Demasiados parámetros. Construya una función para calcular la suma de los divisores de un entero (algoritmo visto en la página 277)

```
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;  
}
```



Llamada:

```
resultado = SumaDivisores(10); // Resultado siempre correcto
```

Una versión con demasiados parámetros podría ser la siguiente:

```
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;  
}
```



Llamada:

```
resultado = SumaDivisores(10, 5); // Resultado correcto  
resultado = SumaDivisores(10, 4); // Resultado incorrecto
```

Ejemplo. Demasiados parámetros:

Re-escribimos el ejemplo del MCD de la página 312. 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 312, 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(primer, segundo);  
        mcd_encontrado = false;  
  
        while (!mcd_encontrado){  
            if (primero % divisor == 0 && segundo % divisor == 0)  
                mcd_encontrado = true;  
            else  
                divisor--;  
        }  
        mcd = divisor;  
    }  
  
    return mcd;  
}  
  
int main(){
```



```
int un_entero, otro_entero, maximo_comun_divisor;

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

maximo_comun_divisor = MCD(un_entero, otro_entero);

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

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

Los ejemplos anteriores ponen de manifiesto lo siguiente:

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.

III.6.3. Precondiciones

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

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

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

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

¿Debemos comprobar dentro de la función si se satisfacen sus precondiciones? Y si lo hacemos, ¿qué acción debemos realizar en caso de que no se cumplan? Contestaremos con detalle más adelante.

III.6.4. Documentación de una función

Hay dos tipos de comentarios:

- *Descripción del algoritmo que implementa la función*

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

Ejemplo. El mayor de tres números

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

En el examen es imperativo incluir la descripción del algoritmo.

IMPORTANT

► Descripción de la cabecera de la función

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

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

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

Incluiremos:

- ▷ Una descripción breve del cometido de la función.

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



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

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

Ampliación:

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



Tema IV

Registros, vectores y matrices

Objetivos:

- ▷ Introducir las primeras *estructuras de datos* que permiten almacenar un conjunto de datos bajo un mismo nombre.
- ▷ Introducir los registros (*struct*) para gestionar conjuntos de datos de distinto tipo y accesibles por nombre.
- ▷ Introducir los vectores y matrices de C++ para gestionar conjuntamente datos del mismo tipo, accesibles mediante índice(s).
- ▷ Presentar los algoritmos básicos de búsqueda y ordenación.
- ▷ Introduciremos las matrices (vectores de varias dimensiones).
- ▷ Informar sobre la variedad de situaciones que podemos encontrar cuando los datos se organizan en matrices y presentar algunas soluciones.

Autor: Juan Carlos Cubero.

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

IV.1. Registros

IV.1.1. El tipo de dato struct

Un *registro* o *struct* permite almacenar varios elementos de (posiblemente) distintos tipos y gestionarlos como uno sólo. Cada uno de los datos agrupados en un *struct* es un *campo*.

Los campos de un *struct* se suponen relacionados, que hacen referencia a una misma entidad. Cada campo tiene un *nombre*.

Un ejemplo típico es la representación en memoria de un *punto* en un espacio bidimensional. Un punto está caracterizado por dos valores: abscisa y ordenada. En este caso, tanto abscisa como ordenada son del mismo tipo, supongamos que sea *double*.

```
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 *un_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 iniciarse 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.1.2. struct y funciones

Un dato struct puede pasarse como parámetro a una función y una función puede devolver un struct.

Por ejemplo, la siguiente función recibe dos struct y devuelve otro.

```
Punto2D 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);  
}
```

La función recibe dos puntos (dos datos struct de tipo Punto2D) y calcula y devuelve el punto medio entre ambos.

- ▷ Los valores recibidos se gestionan en los parámetros formales punto1 y punto2.
- ▷ El resultado se calcula sobre la variable local punto_medio.

- ▷ Observe que el cálculo de `punto_medio` se hace considerando cada componente (`punto_medio.abscisa` y `punto_medio.ordenada`) por separado. Se calculan como la semisuma de las coordenadas apropiadas. Observe cómo se emplea el operador `punto` para acceder al campo apropiado.
- ▷ Finalmente se devuelve una copia del valor de `punto_medio`.

IV.1.2.1. NOTA: Funciones que devuelven struct

Una función en C++ devuelve (return) un único dato. Esta restricción de C++ puede “saltarse” devolviendo un struct que actúe como *contenedor*.

Algunas funciones de C++ se comportan de esta manera. Por ejemplo, la función `div`. En los manuales de referencia puede leer que `div` es una función sobrecargada con tres prototipos:

```
div_t div (int numer,int denom);
ldiv_t div (long int numer, long int denom);
lldiv_t div (long long int numer, long long int denom);
```

`div` devuelve el cociente entero y el resto de la división de `numer` por `denom` en un struct de tipo `div_t`, `ldiv_t` ó `lldiv_t` (para su uso hay que incluir `cstdlib`).

Estos tipos struct tienen dos campos: `quot` y `rem` P.e. para el caso de `div_t`:

```
div_t div (int numer,int denom);
```

sus campos están definidos de tipo `int`:

```
int quot; // cociente
int rem; // resto
```

Ejemplo. Este programa parejas de números enteros y calcula su cociente y resto con `div`. Finaliza cuando ante la lectura del dividendo se introduce un cero. Se realiza la prueba de la división.

```
#include <iostream>
#include <cstdlib> // para div y div_t
using namespace std;

int main ()
{
    div_t divresult;
    int dividendo, divisor;

    cout << "Dividendo = ";
    cin >> dividendo;
    cout << "Divisor = ";
    cin >> divisor;

    divresult = div (dividendo, divisor);

    cout << dividendo << " div " << divisor << " => " << endl;
    cout << '\t' << "Cociente = " << divresult.quot << endl;
    cout << '\t' << "Resto =      " << divresult.rem << endl;

    if (dividendo == divisor*divresult.quot + divresult.rem) {
        cout << "Perfecto --> ";
        cout << dividendo << " = (" << divisor << " * " << divresult.quot
            << ") + " << divresult.rem << endl;
    }
    else cout << "Opps... problemas al dividir." << endl;
}

return 0;
```

Como puede observar, puede resultar útil disponer de estos tipos contenedores `div_t`, `ldiv_t` ó `lldiv_t` ya que con una llamada a `div` obtenemos los dos valores buscados. Otras funciones de C++ ofrecen `struct` contenedores muy útiles (por ejemplo, el tipo `tm` de `ctime`, aunque queda fuera del alcance de esta asignatura).

Podríamos estar tentados de crear tipos contenedores con el objetivo de devolver varios valores desde una función, con la misma filosofía que los tipos mencionados. Será posible (y acertado) si lo devuelto puede verse como una *unidad lógica*, es decir, si los componentes del `struct` son propiedades de una misma entidad o están fuertemente relacionados.

Nunca debería definirse un `struct` con el objetivo de disponer de un *cajón de sastre*, que normalmente es el resultado de una función mal diseñada (habitualmente porque realiza más de una tarea).

Veamos un ejemplo de un mal diseño. Revise la función `MCD` (página 351) que calcula el máximo común divisor de dos valores enteros, con cabecera:

```
int MCD (int primero, int segundo);
```

La función calcula, internamente, el mínimo entre `primero` y `segundo` así que ¿por qué no aprovechar ese cálculo en la función que hace la llamada a `MCD` si nos va a hacer falta? Habría que:

1. Crear un tipo `struct` (llamado p.e. Pareja) para que contuviera el máximo común divisor y el mínimo entre dos enteros.
2. Modificar la función `MCD` para que devolviera un dato de tipo `Pareja`

```
struct Pareja {  
    int mcd;  
    int menor;  
};  
Pareja MCD (int primero, int segundo);
```

Con esta modificación:

1. Obligamos a acompañar siempre que se use al tipo pareja.
2. Cuando queramos calcular únicamente el máximo común divisor únicamente, tendremos que declarar un dato de tipo Pareja que recibirá el resultado de ejecutar la función MCD y sólo se empleará el campo que contiene el máximo común divisor. Esto parecerá un poco extraño e innecesario.

En definitiva, hemos construido una función *multiuso* que devuelve valores cuya relación es difícilmente justificable. Esta manera de trabajar, que parece forzada, es muy habitual entre los programadores noveles.

Puestos a forzar la máquina piense en necesite calcular el máximo común divisor y el mínimo común múltiplo de dos valores enteros. Suponga que dispone de las funciones:

```
int MCD (int primero, int segundo);  
int MCM (int primero, int segundo);
```

Para asegurar la máxima independencia de las funciones involucradas podría:

1. Crear un tipo struct (p.e. Multiplicidad) para contener el máximo común divisor y el mínimo común múltiplo entre dos enteros.

```
struct Multiplicidad {  
    int mcd; // Máximo común divisor  
    int mcm; // Mínimo común múltiplo  
};
```

2. Escribir la función:

```
Multiplicidad MCD_MCM (int primero, int segundo);
```

que llamará a las funciones MCD y MCM, funciones que podrán ser usadas independientemente y de la manera prevista.

IV.1.3. 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 variables globales 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 declarara 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.1.4. 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`:

```
struct Persona {  
    string nombre;  
    string apellidos;  
    string NIF;  
    char categoria;  
    double salario_bruto;  
};
```

podríamos escribir:

```
Persona una_persona = {"", "", "", 'A', 0.0};
```

que asigna, por orden, 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. Vectores

IV.2.1. Introducción

IV.2.1.1. 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

IV.2.1.2. 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 a priori 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++)

No puede declararse un vector con un tamaño variable

```
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];
```

notas =

?	?	?
---	---	---

Otros ejemplos:

```
int main(){  
    const int NUM_REACTORES = 20;  
  
    int TemperaturasReactores[NUM_REACTORES]; // Correcto.  
    bool casados[40]; // Correcto.  
    char NIF[8]; // Correcto.  
    .....  
}
```

IV.2.2. Operaciones básicas

IV.2.2.1. Acceso

Dada la declaración:

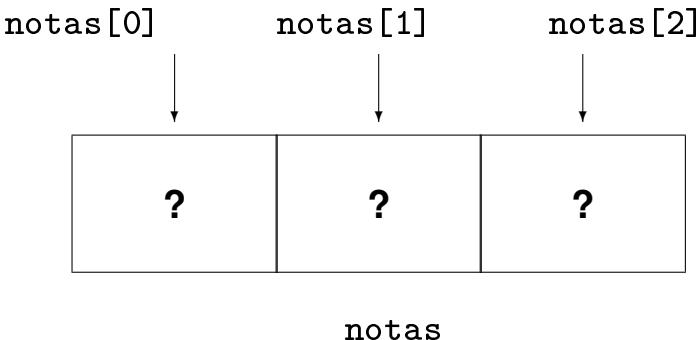
```
<tipo> <identificador> [<núm. componentes>];
```

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 $<\text{núm. componentes}> - 1$

```
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)* .
- ▷ 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`, pasárlas 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.

IV.2.2.2. 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];  
  
    notas[0] = 5.3;  
  
    int i = 0;  
    notas[i] = 5.3;  
  
    notas[0]      notas[1]      notas[2]  
    ↓          ↓          ↓  
    5.3       ?         ?  
    |         |         |  
    notas
```

```
int main(){
    const int MAX_ALUMNOS = 3;
    double notas[MAX_ALUMNOS];
    double unaNota;

    notas[0] = 5.3;           // Correcto. double = double
    notas[1] = 7;             // Correcto. double = int
    unaNota = notas[1];       // Correcto. double = double
```

IV.2.2.3. 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;
```

IV.2.2.4. Inicialización

C++ permite inicializar una variable vector en la declaración.

```
int vector[3] = {4,5,6};
```

inicializa vector[0]=4, vector[1]=5, vector[2]=6

```
int vector[7] = {8};
```

inicializa la primera a 8 y el resto a cero.

```
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};
```

IV.2.3. Representación en memoria

¿Qué ocurre si accedemos a una componente fuera de rango?

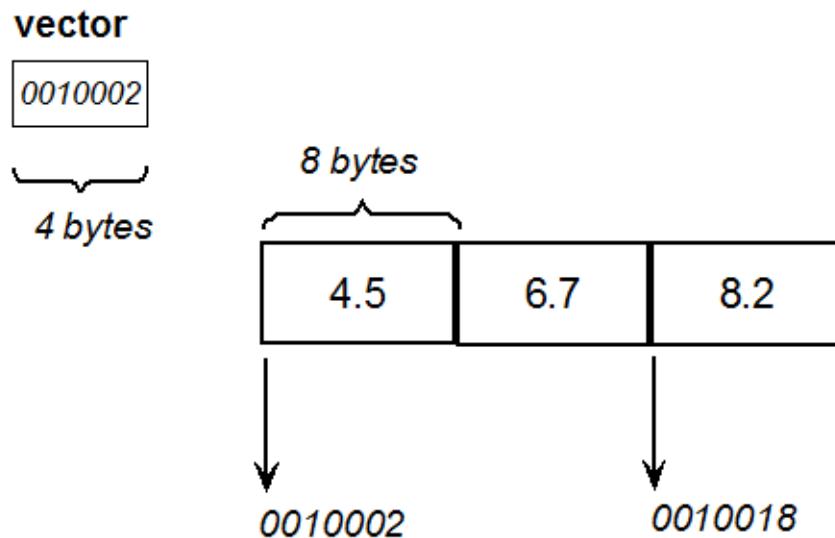
```
cout << notas[15];      // Error lógico. Imprime basura
notas[15] = 5.3;        // Posible error grave en ejecución!
```



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;
```



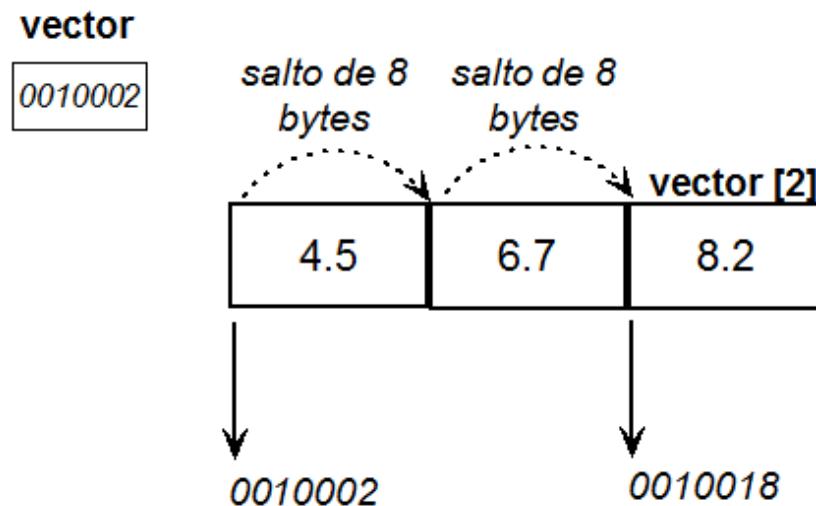
Es como si el programador hiciese la asignación siguiente:

```
vector = 0010002;
```

Realmente, dicha asignación es realizada por el compilador que declara 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:

$$\text{vector}[\text{indice}] \equiv \text{vector} + \text{indice} * \text{sizeof}(<\text{tipo base}>)$$

```
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 modificación de una componente con un índice fuera de rango tiene consecuencias imprevisibles.

IV.2.4. Gestión de componentes utilizadas

Dado un vector, ¿cómo gestionamos el uso de sólo una parte del mismo?

Supongamos el vector `notas` dimensionado para 50 alumnos.

```
const int MAX_ALUMNOS = 50;  
double notas[MAX_ALUMNOS];
```

Si sólo queremos usar parte de él, nos preguntamos dónde debemos colocar los huecos.

Primera opción. Dejamos todos los huecos juntos (normalmente en la zona de índices altos) En este caso, basta usar una variable entera, `util` que indique el número de componentes usadas.

A partir de ahora X denotará una componente utilizada con un valor concreto. El caso anterior sería por tanto del tipo:

```
util = 4  


|     |     |     |     |   |   |     |   |
|-----|-----|-----|-----|---|---|-----|---|
| $X$ | $X$ | $X$ | $X$ | ? | ? | ... | ? |
|-----|-----|-----|-----|---|---|-----|---|


```

Los índices de las componentes utilizadas irán desde 0 hasta $util - 1$:

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

En el ejemplo de las notas:

```
util_notas = 3;  
  
for (int i = 0; i < util_notas; i++)  
    suma = suma + notas[i];
```

Segunda opción. En cualquier sitio:

?	9.5	4.6	?	?	7.8	...	?
---	-----	-----	---	---	-----	-----	---

?	X	X	?	?	X	...	?
---	---	---	---	---	---	-----	---

¿Cómo recorremos el vector? En el ejemplo de las notas:

```
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 (?)

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];
}
```

Siempre que podamos, elegiremos la primera opción, para evitar trabajar con dos vectores dependientes entre sí (el vector que contiene los datos y el que indica los elementos utilizados).

Supongamos que podemos elegir un valor especial (**T**) del tipo de dato de las componentes para indicar que la componente no está siendo usada. Por ejemplo, en un vector de string **T** podría ser la cadena "", o -1 en un vector de positivos.

Entonces tendríamos las siguientes opciones:

Primera opción. Dejar huecos en cualquier sitio y marcarlos con T

T	X	X	T	T	X	...	T
----------	----------	----------	----------	----------	----------	------------	----------

El recorrido será del tipo:

```
for (int i=0; i < MAX; i++){
    if (vector[i] != T){
        .....
    }
}
```

En el ejemplo de las notas:

```
const double NULO = -1; // -1 es una nota IMPOSIBLE

for (int i = 0; i < MAX_ALUMNOS; i++)
    if (notas[i] != NULO)
        suma = suma + notas[i];
```

Segunda opción. Los huecos se dejan al final (o al principio). Ponemos T en la última componente utilizada (debemos reservar siempre una componente para T).

X	X	X	X	T	?	...	?
---	---	---	---	---	---	-----	---

El recorrido será del tipo:

```
for (int i=0; vector[i] != T ; i++){  
    ....  
}
```

En el ejemplo de las notas:

```
const double FIN = -1;  
int i = 0;  
  
while (notas[i] != FIN){  
    suma = suma + notas[i];  
    i++;  
}
```

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 terminador 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 LeeEnteroRango(int min, int max){
    int entero;

    do{
        cin >> entero;
    }while (entero < min || entero > max);

    return entero;
}

int main(){
    const int MAX_ALUMNOS = 100;
    int util_notas;
    double notas[MAX_ALUMNOS], media;
    int superan_media;
```

```
cout << "Introduzca el número de alumnos (entre 1 y " +
        to_string(MAX_ALUMNOS) + "): ";
util_notas = LeeEnteroRango(0, 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 << endl;
cout << superan_media << " alumnos sobre la media";
cout << endl;

return(0);
}
```

http://decsai.ugr.es/~carlos/FP/IV_notas_main.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 $T = ""$

"Pedro"	"Juan"	""	"Carlos"	""	""	...	"María"
---------	--------	----	----------	----	----	-----	---------

X	X	T	X	T	T	...	X
---	---	---	---	---	---	-----	---

```

int main(){
    const string TERMINADOR = "-";
    const string VACIO = "";
    const int MAX_PLAZAS = 50;
    string conductor, nombre_pasajero;
    string pasajeros[MAX_PLAZAS];           // C++ inicializa los string a ""
    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. La función getline permitirá leer datos de tipo string sin esta restricción.

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

En resumen, los tipos más comunes de gestión de un vector son:

▷ **Dejando huecos entre las componentes:**

- Si podemos elegir un valor especial T que represente componente no utilizada.

T	X	X	T	T	X	...	T
---	---	---	---	---	---	-----	---

Los recorridos deben comprobar si la componente actual es T

- Si no podemos elegir dicho valor T, 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):**

- Si podemos elegir un valor especial T que represente componente no utilizada.

Ponemos T en la última componente utilizada (debemos reservar siempre una componente para T).

X	X	X	X	T	?	...	?
---	---	---	---	---	---	-----	---

- Si no podemos elegir dicho valor T, usamos una variable que marque el final:

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 los huecos.

IV.3. Recorridos sobre vectores

En lo que sigue, asumiremos un vector sin huecos y sin destacar ninguna componente T como terminador.

util = 4

X	X	X	X	?	?	...	?
---	---	---	---	---	---	-----	---

Sin pérdida de generalidad, trabajaremos sobre un vector de char.

IV.3.1. Algoritmos de búsqueda

Al proceso de encontrar un elemento específico en un vector se denomina **búsqueda**.

- ▷ **Búsqueda secuencial (Linear/Sequential search)** .
La “técnica” más sencilla. No requiere de ninguna condición previa acerca de los elementos del vector.
- ▷ **Búsqueda binaria (Binary search)** .
Técnica más eficiente. Requiere que el vector esté ordenado.

IV.3.1.1. 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];
.....
vector[0] = 'h'; vector[1] = 'o'; vector[2] = 'l'; vector[3] = 'a';
total_utilizados = 4;

i = 0;
pos_encontrado = -1;

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/~carlos/FP/IV_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.

IV.3.1.2. 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.

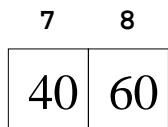
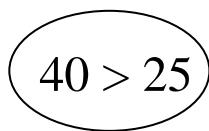
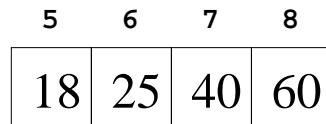
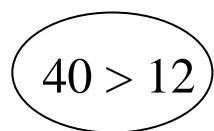
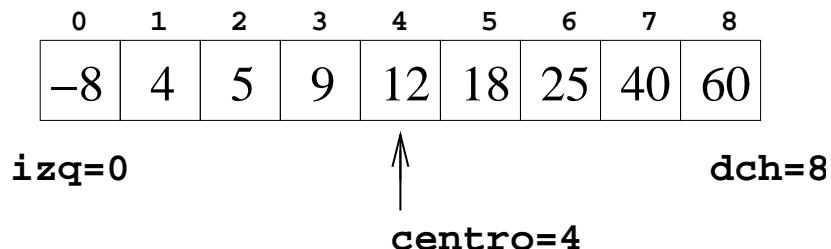
Si coinciden, se habrá encontrado el elemento.

En otro caso, se determina la mitad del vector en la que puede encontrarse.

Se repite el proceso con la mitad correspondiente.

Nota. En el caso de que el valor a buscar estuviese repetido, no se garantiza que se encuentre la primera posición.

Ejemplo. Usamos un vector de enteros para ilustrar el proceso. Queremos buscar el 40



```
int main(){
    const int TAMANIO = 50;
    char vector[TAMANIO];
    .....
    vector[0] = 'a'; vector[1] = 'b'; vector[2] = 'c'; vector[3] = 'd';
    total_utilizados = 4;

    izda = 0;
    dcha = total_utilizados - 1;
    centro = (izda + dcha) / 2;

    while (izda <= dcha && !encontrado){
        if (vector[centro] == buscado)
            encontrado = true;
        else if (buscado < vector[centro])
            dcha = centro - 1;
        else
            izda = centro + 1;

        centro = (izda + dcha) / 2;
    }

    if (encontrado)
        pos_encontrado = centro;
    else
        pos_encontrado = -1;
    .....
```

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

IV.3.1.3. Otras búsquedas

Ejemplo. Encuentre el mínimo elemento de un vector.

v = (h, b, c, f, d, ?, ?, ?, ?)

Algoritmo: Encontrar el mínimo

Iniciar minimo 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;
```

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

Ejemplo. Encuentre un vector dentro de otro.

```
v = (h,b,a,a,a,b,f,d,?,?,?)  (utilizados_v = 8)
a_buscar = (a,a,b,?,?,?,?,?,?,?,?)  (utilizados_a_buscar = 3)
pos_encontrado = 3
```

Algoritmo: Buscar un vector (a_buscar) dentro de otro (v)

Si a_buscar cabe en v

 Recorrer v -desde inicio- hasta que:

- a_buscar no quepa en lo que queda de v
- se haya encontrado a_buscar

 Recorrer las componentes de a_buscar -i- comparando
 a_buscar[i] con v[inicio + i] hasta:

- llegar al final de a_buscar (se ha encontrado)
- encontrar una discrepancia

La búsqueda tendrá éxito cuando se encuentra el contenido de a_buscar en v (los mismos caracteres -todos- en el mismo orden).

En el código que implementa este algoritmo se usa la variable encontrado para indicar si se tiene éxito en la búsqueda.

```
bool encontrado = false; // true si tiene éxito la búsqueda
int inicio = 0;

// inicio indica la posición inicial de v desde donde
// se empieza a buscar la coincidencia con a_buscar.

while (inicio+utilizados_a_buscar<=utilizados_v && !encontrado) {

    bool continuo = true;
    int i = 0;

    // continuo indica si se sigue comparando v (desde inicio)
    // y a_buscar. Mientras hayan coincidencias, sigue iterando.

    while (i < utilizados_a_buscar && continuo)

        if (v[inicio + i] == a_buscar[i]) i++;
        else continuo = false;

    // Si continuo es true, se exploró completamente a_buscar sin
    // encontrar discrepancias con v desde inicio -> ÉXITO.
    // Si continuo es false (cambió su valor inicial), se encontró
    // una discrepancia entre v y a_buscar -> FRACASO
    // (desde inicio no hay coincidencia, no seguir buscando).
    // En este caso a_buscar podría estar más adelante en v, así
    // que habrá que volver a intentarlo -> adelantar inicio.

    if (continuo) { // Exito
        encontrado = true;
        pos_encontrado = inicio;
    } else inicio++;

} // while(inicio+utilizados_a_buscar<=utilizados_v && !encontrado)
```

La conclusión es que si encontrado es true, en la posición pos_encontrado de v empieza una copia del vector a_buscar. Si encontrado es false, entonces a_buscar no está en v.

IV.3.2. Recorridos que modifican componentes

IV.3.2.1. 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



IV.3.2.2. 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**

La componente se *marca* como borrada y habrá que tenerlo en cuenta en el procesamiento posterior.

Es el tipo de borrado usual cuando la ocupación es del tipo:

T	X	X	T	T	X	...	T
---	---	---	---	---	---	-----	---

Obviamente, la ventaja es la eficiencia. 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**

Todas las componentes que hay a la derecha se desplazan una posición a la izquierda.

Es el tipo de borrado usual cuando la ocupación es del tipo que nos ocupa:

util = 4

X	X	X	X	?	?	...	?
---	---	---	---	---	---	-----	---

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.

IV.3.3. 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.

Idea común a algunos algoritmos de ordenación:

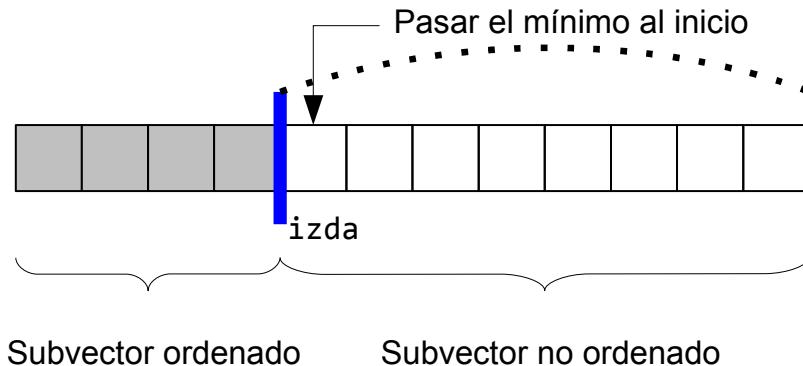
- ▷ El vector se dividirá en dos sub-vectores. El de la izquierda, contendrá componentes ordenadas. Las del sub-vector derecho no están ordenadas.
- ▷ Se irán cogiendo componentes del sub-vector derecho y se colocarán adecuadamente en el sub-vector izquierdo.

Es muy importante conocer los métodos de ordenación para el examen de FP

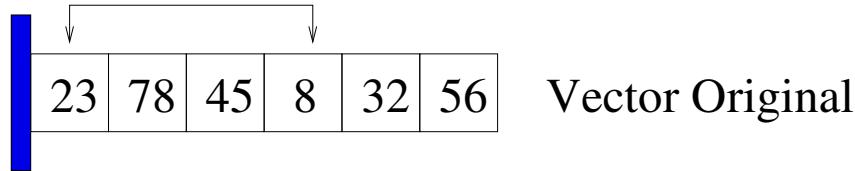
IMPORTANT

IV.3.3.1. Ordenación por Selección

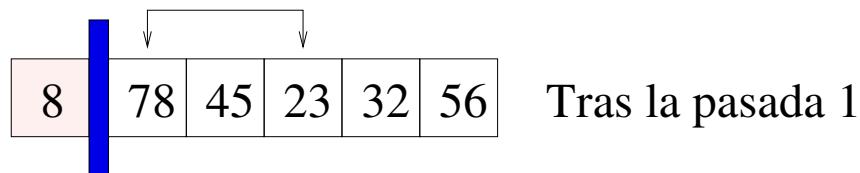
Ordenación por selección (Selection sort) : En cada iteración, se selecciona la componente más pequeña del sub-vector derecho y se coloca al final del sub-vector izquierdo.



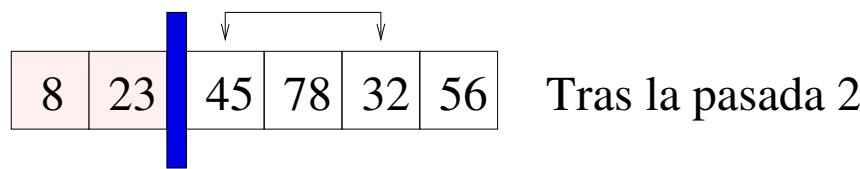
Para ilustrar los métodos usaremos una secuencia de enteros (en cualquier caso, recuerde que una secuencia de `char` no es sino una secuencia de los enteros que representan los órdenes correspondientes)



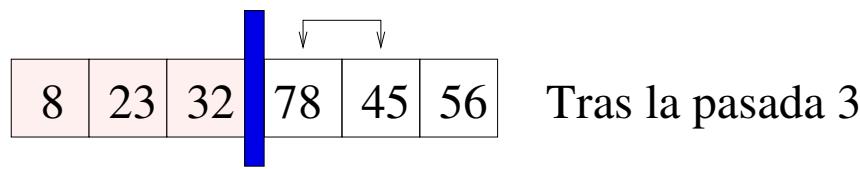
Vector Original



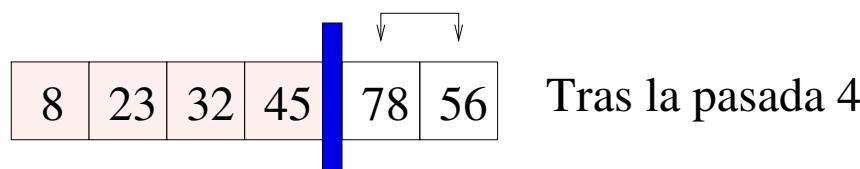
Tras la pasada 1



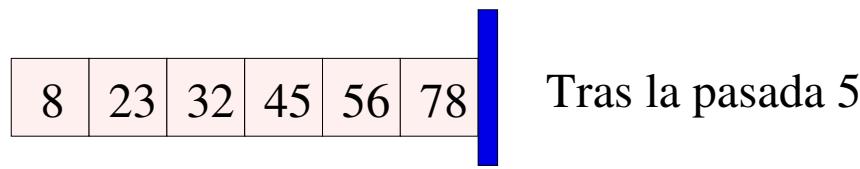
Tras la pasada 2



Tras la pasada 3



Tras la pasada 4



Tras la pasada 5

Algoritmo: Ordenación por 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++){

    // Calcular el mínimo entre "izda" y "total_utilizados"-1

    int minimo = vector[izda]; // Valor del mínimo
    int pos_min = izda;           // Posición del mínimo

    for (int i = izda + 1; i < total_utilizados ; i++)
        if (vector[i] < minimo){ // Nuevo mínimo
            minimo = vector[i];
            pos_min = i;
        }

    // Intercambiar los valores guardados en "izda" y "pos_min"
    intercambia = vector[izda];
    vector[izda] = vector[pos_min];
    vector[pos_min] = 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)



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

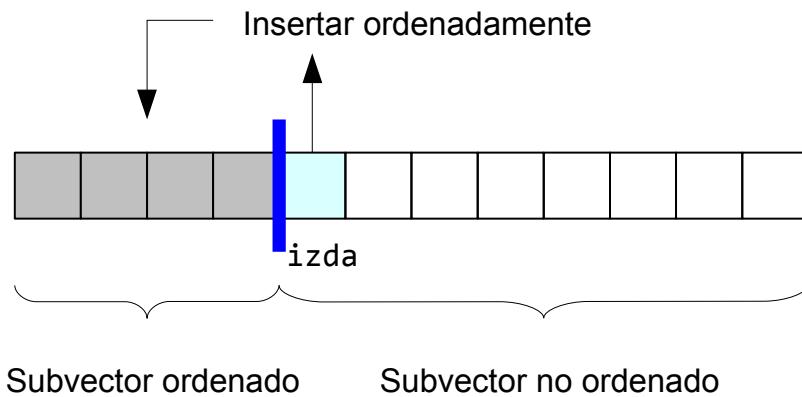
Nota. Estas pruebas también son aplicables al resto de algoritmos de ordenación

Applet de demostración del funcionamiento:

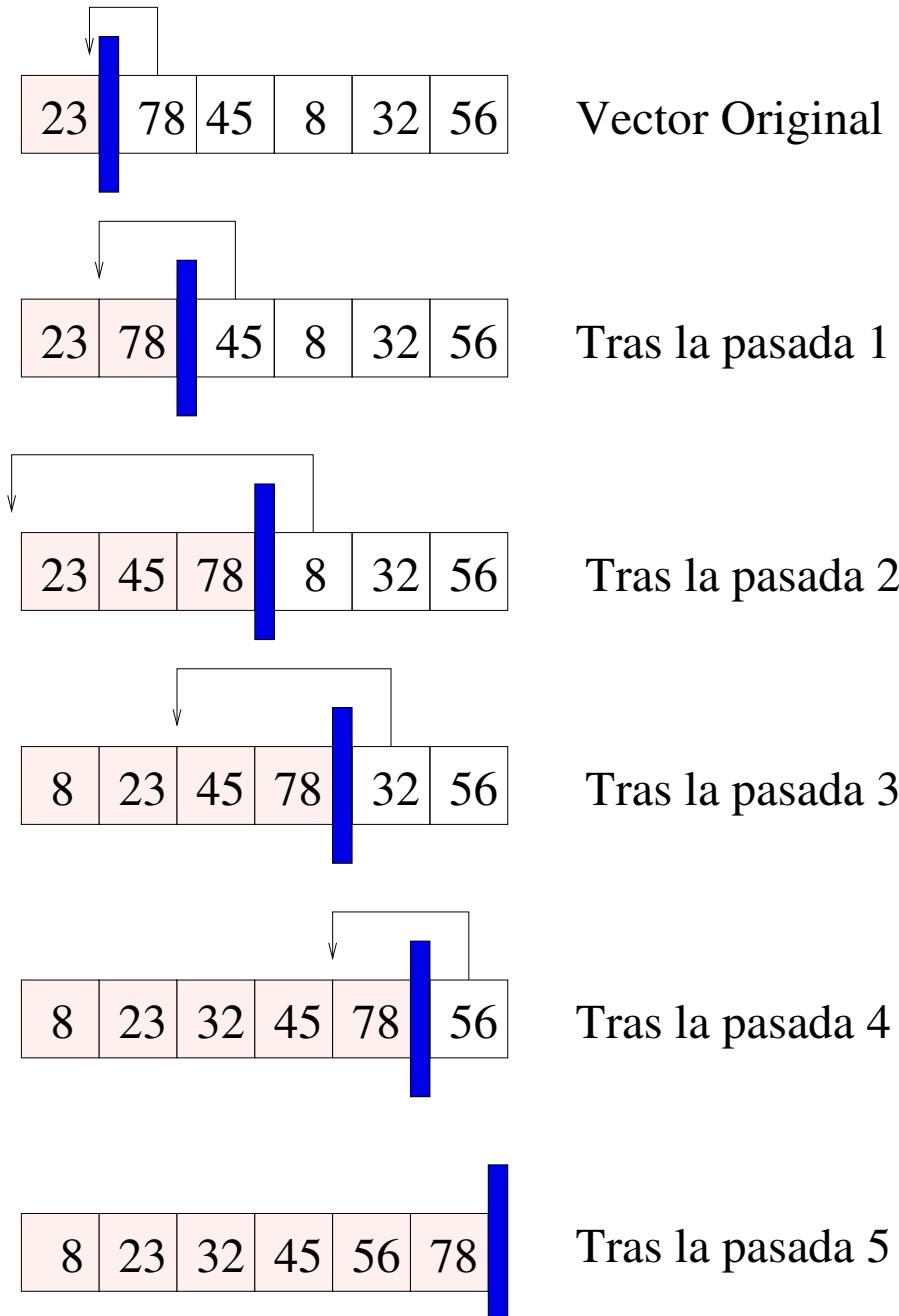
<http://www.sorting-algorithms.com/>

IV.3.3.2. 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)



Algoritmo: Ordenación por Inserción

Ir fijando el inicio del subvector derecho
con un índice izda (desde 1 hasta total_utilizados - 1)
Seleccionar el valor v[izda]
Insertar dicho valor de forma ordenada
en el subvector izquierdo

```
for (int izda = 1; izda < total_utilizados; izda++){

    // "a_insertar" es el valor que se va a insertar en
    // subvector izquierdo. Este subvector está ordenado y
    // comprende las posiciones entre 0 e "izda"-1

    int a_insertar = v[izda];

    // Se busca la posición en la zona ordenada

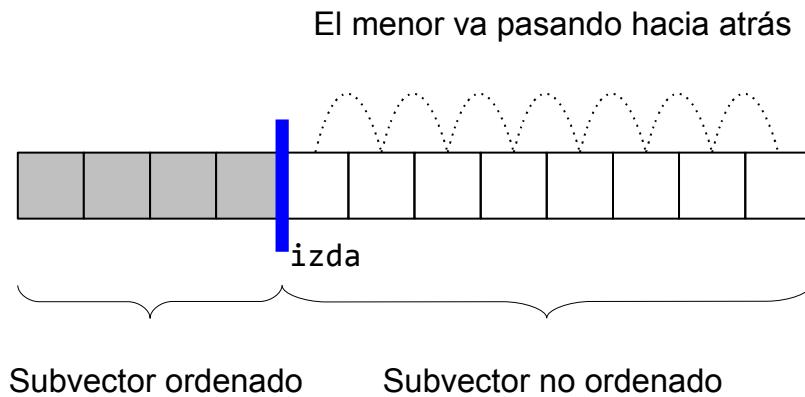
    int i = izda;
    while ((i > 0) && (a_insertar < v[i-1])) {

        v[i] = v[i-1]; // Desplazar a la derecha los
                        // valores mayores que "a_insertar"
        i--;
    }

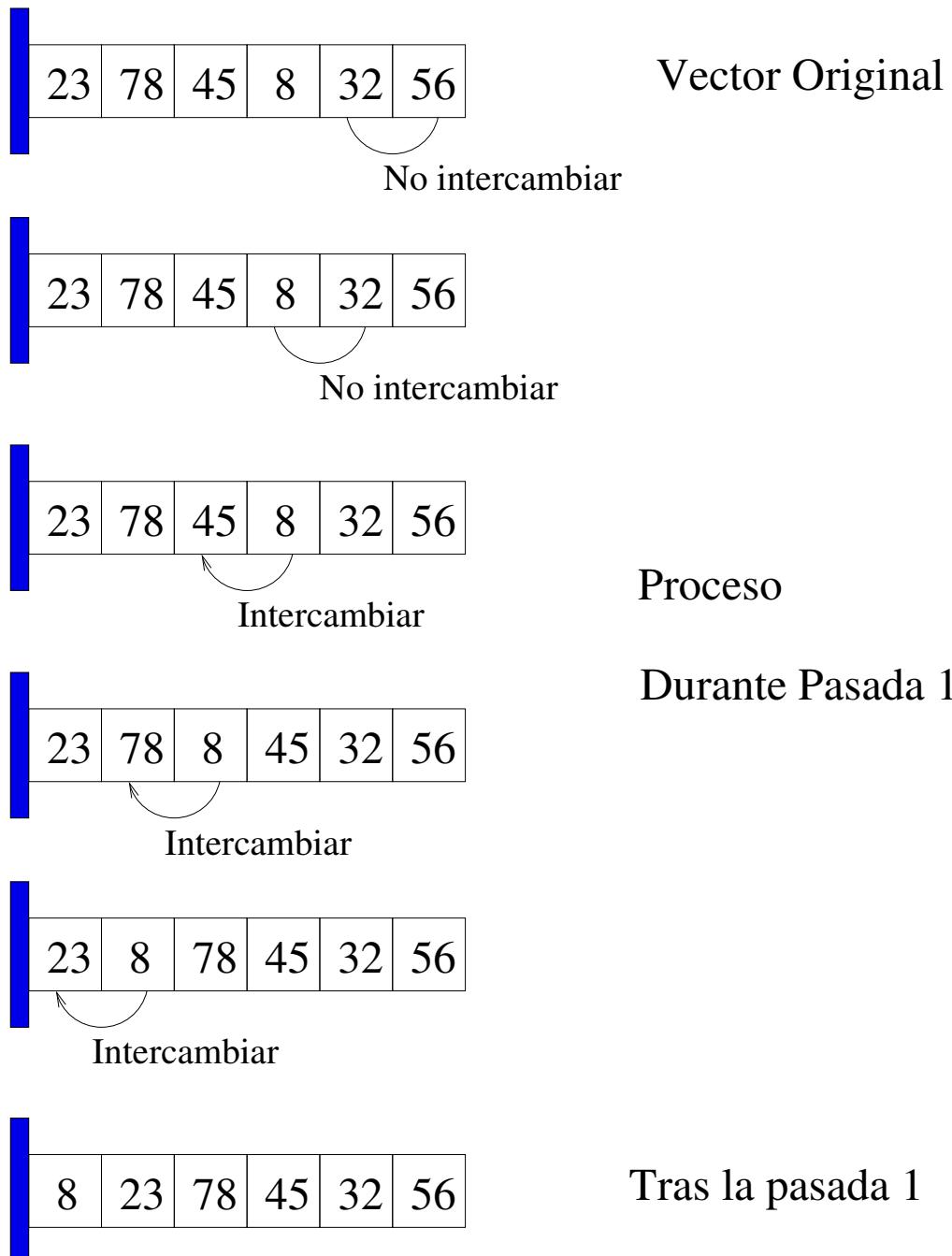
    v[i] = a_insertar; // Copiar -insertar- en el hueco
}
```

IV.3.3.3. Ordenación por Intercambio Directo (Método de la Burbuja)

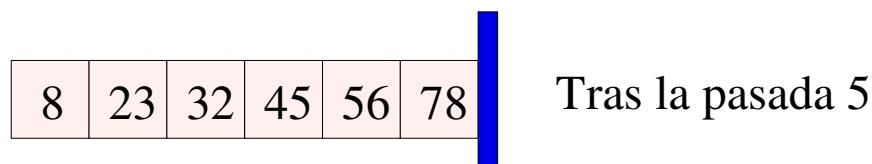
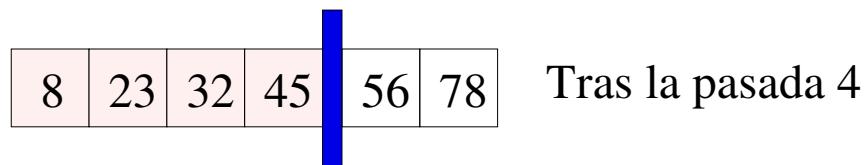
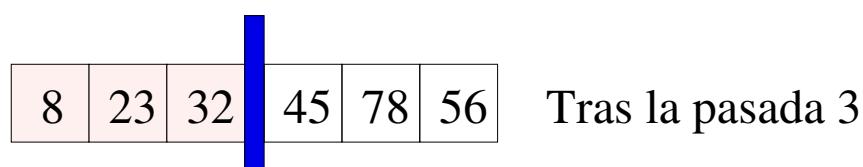
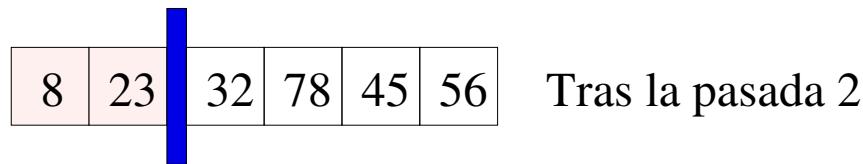
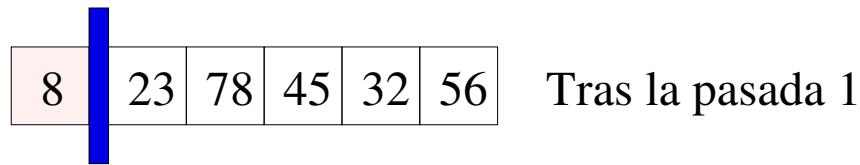
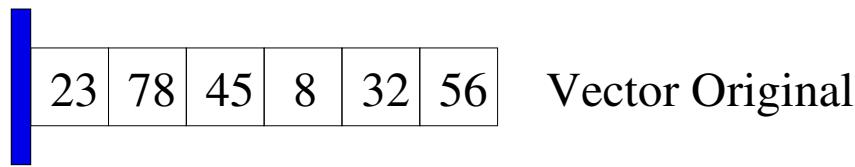
Ordenación por intercambio directo (burbuja) (Bubble sort) : Al igual que antes, a la izquierda se va dejando un subvector ordenado. Desde el final y hacia atrás, se van comparando elementos dos a dos y se deja a la izquierda el más pequeño (intercambiándolos)



Primera Pasada



Resto de Pasadas



Algoritmo: Ordenación por Burbuja

```
Ir fijando el inicio del subvector derecho  
con un contador izda desde 0 hasta total_utilizados - 1  
    Recorrer el subvector de la derecha desde  
        el final hasta el principio (izda)  
        con un contador 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]){ // Intercambiar  
  
            int intercambia = vector[i];  
            vector[i] = vector[i-1];  
            vector[i-1] = intercambia;  
        }  
  
}
```

Mejora posible. Si en una pasada del bucle más interno no se produce ningún intercambio, el vector ya está ordenado. Deberíamos finalizar.

► Segunda Aproximación

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, y finalizamos si se da el caso.

```
bool cambio = true; // Para entrar

for (int izda = 0; izda < total_utilizados && cambio; izda++){

    // En cada pasada iniciamos "cambio" a false.
    // Se pondrá true si y solo si hay algún intercambio

    cambio = false;

    for (int i = total_utilizados-1 ; i > izda ; i--)

        if (vector[i] < vector[i-1]){ // Intercambiar

            int intercambia = vector[i];
            vector[i] = vector[i-1];
            vector[i-1] = intercambia;

            cambio = true; // Se ha hecho un intercambio
        }

}
```

IV.4. Matrices

IV.4.1. Declaración y operaciones con matrices

IV.4.1.1. 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.

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];
    ....
```

IV.4.1.2. 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.

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 ejecución: Fila 2 no existe.
    parcela[0] [3] = 7.2; // Error ejecución: Columna 3 no existe.
    parcela[0] [0] = 4; // Correcto. Casting automático: double = int
```

IV.4.1.3. 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
```

IV.4.2. Representación en memoria (Ampliación)

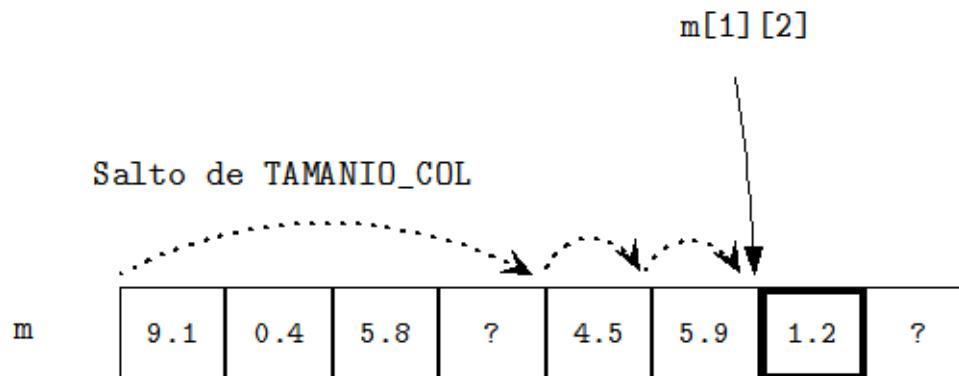
Todas las posiciones de una matriz están realmente contiguas en memoria. La representación exacta depende del lenguaje. En C++ se hace por filas:

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*TAMANIO_COL + j

IV.4.3. Matrices con 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 $2 \times 3 \times 5 = 30$ datos double.

IV.4.4. 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.

IV.4.4.1. Se usan todas las componentes

Ocupamos todas las componentes reservadas. Las casillas no utilizadas tendrán un valor especial del tipo de dato de la matriz.

`MAX_FIL = 15, MAX_COL = 20`

T	X	T	T	X	T	...	X
X	X	T	T	T	X	...	T
T	T	X	X	X	T	...	T
...
X	T	X	T	X	T	...	X

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/~carlos/FP/IV_crucigrama_base.cpp

IV.4.4.2. 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/~carlos/FP/IV_notas_base.cpp

IV.4.4.3. Se ocupa un bloque rectangular

Este tipo de ocupación es bastante usual.

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.

```
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";
    }
}
```

Una palabra viene delimitada por el inicio de la fila, el final de ésta y también por el terminador #. Sólo consideramos palabras en horizontal y de izquierda a derecha.

Vamos a construir un vector con todas las palabras del crucigrama. Cada palabra se almacenará en un string.

Lo vamos a ver de dos formas distintas.

► **Primera forma**

Utilizamos dos apuntadores izda y dcha para delimitar el inicio y final de la palabra. izda se posicionará al inicio de la palabra y dcha irá avanzando hasta el final de la misma. Al llegar al final, construiremos la cadena con los caracteres que haya entre izda y dcha.

Algoritmo: Obtener palabras en horizontal de un crucigrama

Recorrer todas las filas

 Inicializar izda y dcha a cero

 Recorrer la fila actual

 Si el carácter que hay en la columna actual es un separador, avanzar izda y dcha

 si no

 Si el siguiente a dcha es separador o final de fila, estamos al final de una palabra

 => Construir la palabra con los caracteres entre izda y dcha.

 Avanzar izda y dcha al final de la palabra

 si no

 Mantener izda y avanzar dcha

```
string cjto_palabras[MAX_FIL*MAX_COL];  
.....  
util_cjto_palabras = 0;  
  
for (int fil = 0; fil < util_fil; fil++){  
    izda = dcha = 0;  
    hay_columnas = util_col != 0;  
  
    while (hay_columnas){  
        if (crucigrama[fil][izda] == SEPARADOR){  
            izda++;  
            dcha++;  
        }  
        else{  
            siguiente = dcha + 1;  
  
            if (siguiente == util_col ||  
                crucigrama[fil][siguiente] == SEPARADOR){  
  
                for (int j = izda; j <= dcha; j++)  
                    palabra = palabra + crucigrama[fil][j];  
  
                cjto_palabras[util_cjto_palabras] = palabra;  
                util_cjto_palabras++;  
                palabra = "";  
                izda = dcha = siguiente;  
            }  
            else  
                dcha++;  
        }  
        hay_columnas = dcha < util_col;  
    }  
}
```

► Segunda forma

Utilizamos un único apuntador y detectamos las distintas situaciones que se pueden producir.

a indica que hay una letra distinta de separador
indica que hay un separador
... indica que hay cualquier cosa (separador o letra)
! indica la posición actual
| indica el final de la fila

Casos a tener en cuenta:

Casos en los que debo ir construyendo la palabra pero no la he terminado: Cuando el actual no es separador y no he llegado al final

!

...a...|

Casos en los que he terminado una palabra y la añado: Cuando el actual es un separador o final de fila y en cualquier caso, el anterior es una letra

!

...a#...|

!

...a|

Casos en los que no tengo que hacer nada (salvo avanzar, en su caso) Cuando el anterior es un separador o el final de la fila:

!

...##...|

!

...#|

!

#...

Algoritmo: Obtener palabras en horizontal de un crucigrama (2)

Usaremos dos variables:

actual_es_separador

anterior_es_separador (el anterior al actual)

Recorrer todas las filas

 Recorrer la fila actual

 Si no hemos rebasado el final y el actual no es separador

 Concatenar el carácter actual en la palabra
 que se está formando.

 si no

 Si el actual es un separador o final de fila y
 en cualquier caso, el anterior es una letra

 Añadir la palabra que se estaba formando

```
util_cjto_palabras = 0;
anterior_es_separador = false;
actual_es_separador = false;

for (int fil = 0; fil < util_fil; fil++){
    col = 0;
    anterior_es_separador = true;

    do{
        hay_columnas = col < util_col;

        if (hay_columnas)
            actual_es_separador = crucigrama[fil][col] == SEPARADOR;

        if (hay_columnas && ! actual_es_separador){
            palabra = palabra + crucigrama[fil][col];
            anterior_es_separador = false;
        }
        else{
            if (! anterior_es_separador &&
                (!hay_columnas || actual_es_separador)){
                cjto_palabras[util_cjto_palabras] = palabra;
                util_cjto_palabras++;
                palabra = "";
                anterior_es_separador = true;
            }
        }
        col++;
    }while (hay_columnas);
}
```

http://decsai.ugr.es/~carlos/FP/IV_crucigrama_obtener_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 397

Algoritmo: Buscar una palabra en una sopa de letras

Recorrer todas las filas -fila- 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 fila
- se haya encontrado a_buscar

Recorrer las componentes de a_buscar -i- comparando
a_buscar[i] con sopa[fila][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];
bool encontrado, va_coincidiendo;
int fil_encontrado, col_encontrado;
int tamanio_a_buscar;

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 >> tamanio_a_buscar;

for (int i = 0; i < tamanio_a_buscar; i++)
    cin >> a_buscar[i];

encontrado = false;
fil_encontrado = col_encontrado = -1;
```

```
for (int fil = 0; fil < util_fil && !encontrado; fil++) {
    hay_espacio_hasta_el_final = true;

    for (int col_inicio = 0;
        col_inicio + tamanio_a_buscar <= util_col && !encontrado;
        col_inicio++){
        va_coincidiendo = true;

        for (int i = 0; i < tamanio_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/~carlos/FP/IV_sopa.cpp

IV.4.4.4. Se ocupan las primeras filas, pero con tamaños distintos

```
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. Podemos considerar un texto como un conjunto de líneas, siendo cada una un conjunto de caracteres representados en una fila.

```
MAX_LINEAS = 200, MAX_LONG_LINEA = 100
lineas_utilizadas = 2
col_utilizadas_por_fila = {4, 3, ?, ..., ?}
```

h	o	l	a	?	?	...	?
b	y	e	?	?	?	...	?
?	?	?	?	?	?	...	?
...
?	?	?	?	?	?	...	?

```
int main(){
    const int MAX_LINEAS = 200, MAX_LONG_LINEA = 100;
    char frase [MAX_LINEAS][MAX_LONG_LINEA];
    int col_utilizadas_por_fila[MAX_LINEAS];
    int lineas_utilizadas;
```


Tema V

Clases

Objetivos:

- ▷ Introducir el principio de *encapsulación* que permitirá empaquetar datos y funciones en un mismo módulo: la *clase* (class).
- ▷ Profundizar en el principio de *ocultación de información* (ámbitos public y private)
- ▷ Saber crear *objetos* y sus implicaciones: métodos *constructores*.
- ▷ Ver cómo gestiona C++ la *copia de objetos*. Saber distinguir claramente el *constructor de copia* y el *operador de asignación*.
- ▷ Distinguir entre *constantes a nivel de objeto* y *constantes a nivel de clase* (*constantes static*)
- ▷ Gestionar correctamente: vectores como datos miembro, vectores como datos locales de un método, vectores como parámetros a un método y vectores de objetos
- ▷ La clase Secuencia.

Autor: Juan Carlos Cubero.

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

V.1. Motivación. Clases y Objetos

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

- ▷ Principio de una única vez. Identificando tareas y empaquetando las instrucciones que la resuelven.

Si necesitamos realizar la misma tarea de nuevo, llamaremos a la función de nuevo (posiblemente con valores diferentes).

No es importante (relativamente) *cómo* se resuelve la tarea (qué algoritmo se emplea para su resolución). Sólo necesito saber *cómo* debo llamar a la función y qué devuelve.

- ▷ Principio de ocultación de información.

Las funciones pueden emplear sus propios datos locales (además de los parámetros formales) para realizar la tarea encomendada. Esos datos solo se conocen *dentro* de la función, durante su ejecución. De la misma manera, una función no podrá acceder a los datos locales de otra.

Por otra parte, los registros (`struct`) permiten la creación de *nuevos tipos de datos*. Podemos modelar entidades complejas constituidas por varios campos (puntos, polígonos, personas, cuentas bancarias, ...) mediante `struct` y referirnos a ellas por un sólo nombre, pudiendo pasar a funciones (y obtener de ellas) datos de estos tipos.

Los `struct`, no obstante, aunque permiten un mecanismo de acceso restringido a sus campos, no permiten encapsular las funciones que los gestionan, manteniendo la separación entre datos y funciones. Dicho de otra manera, la gestión recae directamente en las funciones que reciben `struct` y/o los devuelven.

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 que permite cumplir los principios de programación:

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

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 y se comunican con el resto del programa a través de sus parámetros (entrada) y del resultado que devuelven (salida).

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

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

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

Programación orientada a objetos (Object oriented programming) (PDO). Metodología de programación modular en la que los módulos o paquetes se denominan *objetos (objects)* .

La base del diseño de una solución a un problema usando PDO consiste en analizar las *entidades* que intervienen en el problema e implementarlas usando *objetos*.

Un objeto aglutina en un único paquete *datos y funciones*.

- ▷ Los datos representan las características de una entidad.
- ▷ Las funciones incluidas en un objeto se denominan *métodos (methods)* . 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)

Nota:

Observe que un objeto es un dato **compuesto** e incluye otros datos y métodos.

Para construir un objeto, primero tenemos que definir su estructura. Esto se hace con el concepto de clase.

Una **clase (class)** es un **tipo de dato** definido por el programador. Se usa para representar una **entidad**.

Con la clase especificamos las características comunes y el comportamiento de una entidad. Es como un patrón o molde a partir del cual construimos los objetos.

Un **objeto (object)** es un **dato** cuyo tipo de dato es una clase. También se dirá que un objeto es la **instancia (instance)** o caso particular de una clase.

```
class MiClase{  
    ....  
};  
int main(){  
    MiClase un_objeto_instancia_de_MiClase;  
    MiClase otro_objeto_instancia_de_MiClase;  
    ....  
}
```

```
class CuentaBancaria{           // <- clase  
    ....  
};  
int main(){  
    CuentaBancaria una_cuenta;    // <- un objeto  
    CuentaBancaria otra_cuenta;   // <- otro objeto  
    ....  
}
```

Los objetos `una_cuenta` y `otra_cuenta` existirán mientras esté ejecutándose `main`.

V.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 empaquetar datos y funciones en un mismo módulo.

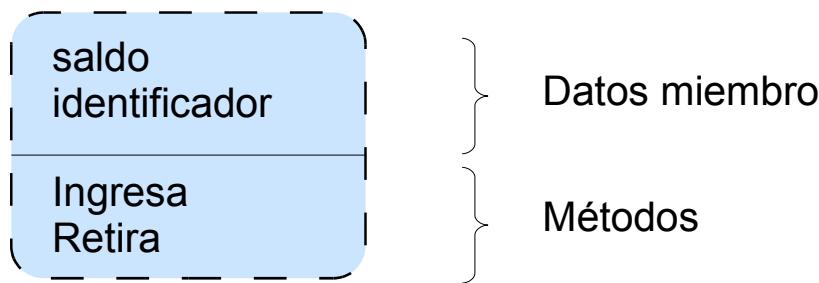
Una clase se compone de:

- ▷ **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.

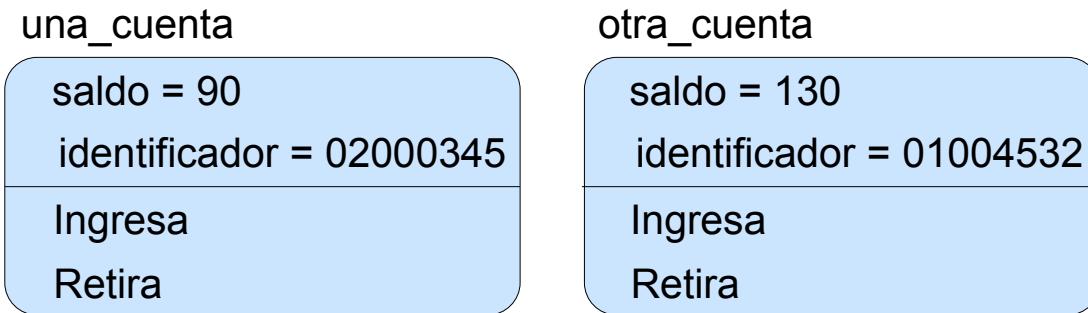
```
class CuentaBancaria{           // <- clase
    ...
};

int main(){
    CuentaBancaria una_cuenta;   // <- un objeto
    CuentaBancaria otra_cuenta;  // <- otro objeto
    ...
}
```

Clase CuentaBancaria:



Objetos instancia de CuentaBancaria:



V.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 analizando el **ámbito público**.

Todas las declaraciones incluidas después de public: son públicas, es decir, accesibles desde fuera del objeto. Desde la función main o desde los métodos de otros objetos accederemos a los datos públicos de los objetos con la sintaxis propia del acceso a los campos de un dato struct: nombre del objeto, un punto y el nombre del dato.

```
class MiClase{  
public:  
    int dato;          // Valor indeterminado  
};  
  
int main(){  
  
    MiClase un_objeto, otro_objeto;  
  
    un_objeto.dato = 4;  
    cout << un_objeto.dato;      // Imprime 4  
    un_objeto.dato = 8;  
    cout << un_objeto.dato;      // Imprime 8  
    otro_objeto.dato = 7;  
    cout << otro_objeto.dato;    // Imprime 7
```

Cada vez que modificamos un dato miembro, diremos que se ha modificado el estado del objeto.

Ejemplo. Cuenta bancaria.

Nota. Esta clase tiene problemas importantes de diseño que se irán arreglando a lo largo de este tema.

```
class CuentaBancaria{  
public:  
    string identificador;      // Cadena vacía, ""  
    double saldo;                // Valor indeterminado, x  
};  
  
int main(){  
    CuentaBancaria cuenta;          // {"", x}  
  
    string identificador_cuenta;  
    double ingreso, retirada;  
  
    cout << "\nIntroduce identificador a asignar a la cuenta:";  
    cin >> identificador_cuenta;  
    cuenta.identificador = identificador_cuenta;  
  
    cout << "\nIntroduce cantidad inicial a ingresar: ";  
    cin >> ingreso;  
  
    cuenta.saldo = ingreso;  
  
    cout << "\nIntroduce cantidad a ingresar: ";  
    cin >> ingreso;  
  
    cuenta.saldo = cuenta.saldo + ingreso;  
  
    cout << "\nIntroduce cantidad a retirar: ";  
    cin >> retirada;
```



```
cuenta.saldo = cuenta.saldo - retirada;

CuentaBancaria otra_cuenta; // Otro objeto de la clase CuentaBancaria

otra_cuenta.saldo = 30000; // <- No afecta al otro objeto cuenta

saldo = 300; // Error de compilación.
// saldo no es un dato definido en main.
```

Algunas aclaraciones:

- ▷ En vez de poner

```
cin >> identificador_cuenta;
cuenta.identificador = identificador_cuenta;
```

podríamos haber puesto directamente:

```
cin >> cuenta.identificador;
```

- ▷ Con las herramientas que conocemos, no puede leerse directamente un objeto por completo:

```
CuentaBancaria cuenta;

cin >> cuenta; // Error de compilación
```

Cuando se crea un objeto, los datos miembro no tienen ningún valor asignado por defecto, a no ser que se inicialicen dentro de la clase.

Desde C++ 11, la inicialización puede realizarse en la definición del dato miembro. Observe la inicialización de saldo e identificador. Cada vez que se declare un objeto CuentaBancaria su campo saldo tomará el valor 0.0 y su campo identificador contendrá la cadena vacía.

```
class CuentaBancaria{  
public:  
    string identificador = "";  
    double saldo = 0.0;  
};  
int main(){  
    CuentaBancaria cuenta; // Inicialización: {"", 0}  
    cout << cuenta.saldo << "\n"; // 0  
    cout << cuenta.identificador; // no se muestra nada (cadena vacía)
```



469

Las inicializaciones especificadas en la declaración de los datos miembro dentro de la clase, se aplican en el momento que se crea un objeto cualquiera de dicha clase.

Nosotros no emplearemos esta inicialización sino que delegaremos esta tarea a los métodos constructores (sección V.4) cuya tarea es la crear e inicializar los objetos.

Nota:

Realmente, C++ inicializa siempre cualquier string a "" en el momento de su declaración, por lo que no es necesario hacerlo explícito.

V.2.2. Métodos

Por ahora, hemos conseguido crear varios objetos (cuentas bancarias) con sus propios datos miembros (saldo, identificador).

Los métodos determinan el comportamiento de los objetos de la clase. Son funciones definidas dentro de la clase que se ejecutan *sobre* los objetos de la clase.

- ▷ El comportamiento de los métodos es el mismo para todos los objetos de la clase.

Como los datos miembro, los métodos podrán ser *públicos* o *privados*. Por ahora sólo consideramos *métodos públicos*.

Desde la función `main` o desde métodos de otros objetos accederemos a los métodos públicos de los objetos con la sintaxis conocida: nombre del objeto, un punto, el nombre del método y entre paréntesis los parámetros (si los hubiera).

```
class MiClase{  
public:  
    int dato;           // valor indeterminado, x  
  
    void UnMetodo() {  
        ....  
    };  
};  
int main(){  
    MiClase un_objeto;    // {x}  
  
    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

Usaremos nombres para denotar las clases y verbos para los métodos.

- ▷ Los métodos pueden modificar el estado del objeto sobre el que actúan, es decir, pueden modificar los datos miembro. Acceden a ellos directamente sin restricciones.

Ejemplo. Añadimos métodos para ingresar y sacar dinero en la cuenta bancaria:

```
class CuentaBancaria{           // <- clase
public:
    string identificador ;     // ""
    double saldo;              // valor indeterminado, x

    void Ingresa(double cantidad){ // Dentro del método accedemos
        saldo = saldo + cantidad; // directamente al dato miembro
    }
    void Retira(double cantidad){ // Dentro del método accedemos
        saldo = saldo - cantidad; // directamente al dato miembro
    }
};

int main(){
    CuentaBancaria una_cuenta;      // {"", x}
    CuentaBancaria otra_cuenta;    // {"", x}

    una_cuenta.saldo = 0;
    una_cuenta.identificador = "20310381450100006529"; // {"2...9", 0}

    una_cuenta.Ingresa(25);          // {"2...9", 25}
    una_cuenta.Retira(10);          // {"2...9", 15}
    .....

    otra_cuenta.saldo = 0;
    otra_cuenta.identificador = "20310381450100007518"; // {"2...8", 0}

    otra_cuenta.Ingresa(45);          // {"2...8", 45}
    otra_cuenta.Retira(15);          // {"2...8", 30}
    .....
```



469

A destacar:

- ▷ **Los métodos** Ingresá y Retira acceden al dato miembro saldo por su nombre.

Los métodos acceden a los datos miembro directamente.

- ▷ **Los métodos** Ingresá y Retira modifican alguno de los datos miembro.

Los métodos pueden modificar el estado del objeto.

Ejercicio. Definamos la clase SegmentoDirigido

```
class SegmentoDirigido{  
public:  
    double x_1, y_1, x_2, y_2; // valores indeterminados, x  
};  
int main(){  
    SegmentoDirigido un_segmento; // {x, x, x, x}  
  
    cout << un_segmento.x_1; // Valor indeterminado  
  
    un_segmento.x_1 = 3.4;  
    un_segmento.y_1 = 5.6;  
    un_segmento.x_2 = 4.5;  
    un_segmento.y_2 = 2.3;  
  
    cout << un_segmento.x_1; // 3.4  
    cout << un_segmento.x_2; // 4.5  
}
```



472

Ejercicio. Defina sendos métodos TrasladaHorizontal y TrasladaVertical para trasladar un segmento un número de unidades.

```
class SegmentoDirigido{  
public:  
    double x_1, y_1; // valores indeterminados, x  
    double x_2, y_2; // valores indeterminados, x  
  
    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; // {x, x, x, x}  
  
    un_segmento.x_1 = 3.4; // {3.4, x, x, x}  
    un_segmento.y_1 = 5.6; // {3.4, 5.6, x, x}  
    un_segmento.x_2 = 4.5; // {3.4, 5.6, 4.5, x}  
    un_segmento.y_2 = 2.3; // {3.4, 5.6, 4.5, 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  
}
```



472

Ejercicio. Calcule la longitud de un segmento dirigido.

```
class SegmentoDirigido{  
public:  
    double x_1, y_1; // valores indeterminados, x  
    double x_2, y_2; // valores indeterminados, x  
  
    double Longitud(){  
        double resta_abscisas = x_2 - x_1;  
        double resta_ordenadas = y_2 - y_1;  
  
        return sqrt(resta_abscisas * resta_abscisas +  
                    resta_ordenadas * resta_ordenadas);  
    }  
  
    void TrasladaHorizontal(double unidades){  
        x_1 = x_1 + unidades;  
        x_2 = x_2 + unidades;  
    }  
  
    void TrasladaVertical(double unidades){  
        y_1 = y_1 + unidades;  
        y_2 = y_2 + unidades;  
    }  
};  
int main(){  
    SegmentoDirigido un_segmento; // {x, x, x, x}  
  
    un_segmento.x_1 = 3.4; // {3.4, x, x, x}  
    un_segmento.y_1 = 5.6; // {3.4, 5.6, x, x}  
    un_segmento.x_2 = 4.5; // {3.4, 5.6, 4.5, x}  
    un_segmento.y_2 = 2.3; // {3.4, 5.6, 4.5, 2.3}  
  
    cout << "\nLongitud del segmento = " << un_segmento.Longitud();
```



Para representar las clases se utiliza una notación gráfica con una caja que contiene el nombre de la clase. A continuación se incluye un bloque con los datos miembro y por último un tercer bloque con los métodos. Los métodos públicos se notarán con un símbolo + (posteriormente se verán los privados)



CuentaBancaria
+ double saldo
+ string identificador
+ void Ingresar(double cantidad)
+ void Retirar(double cantidad)



SegmentoDirigido
+ double x_1
+ double y_1
+ double x_2
+ double y_2
+ double Longitud()
+ void TrasladaHorizontal(double unidades)
+ void TrasladaVertical(double unidades)

V.2.3. Controlando la integridad de los objetos

Debemos recordar que debemos asegurar la correcta ejecución de las funciones, que depende de la validez de los parámetros con los que se realiza la llamada.

Esta comprobación podría hacerse *dentro* de la función, técnica que descartamos (¿qué devuelve una función que ya se está ejecutando cuando detecta que el argumento recibido no es correcto?).

Aconsejamos el uso de *precondiciones* que deben cumplirse cuando se llama a la función y se comprueban antes de la llamada. Evitamos que la función realice estas comprobaciones y aseguramos su correcto desempeño. La contrapartida es que se delega al usuario esta tarea.

Con los métodos de las clases procederemos de la misma manera.

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?

```
class CuentaBancaria{  
public:  
    double saldo;  
    string identificador;  
  
    // PRE: 0 < cantidad  
    void Ingresa(double cantidad){  
        saldo = saldo + cantidad;  
    }  
    // PRE: 0 < cantidad <= saldo  
    void Retira(double cantidad){  
        saldo = saldo - cantidad;  
    }  
};
```

```
int main(){
    CuentaBancaria cuenta;
    double ingreso, retirada;

    cuenta.identificador = "20310381450100006529";      // "2...9", X
    .....

    cin >> ingreso;
    if (ingreso > 0)                                ☺
        cuenta.Ingresa(ingreso);

    cin >> retirada;
    if (retirada > 0 && retirada <= cuenta.saldo)   ☺
        cuenta.Retira(retirada);
```

Cada vez que realicemos un ingreso o retirada de fondos, habría que realizar las anteriores comprobaciones.

Si la llamada a un método conlleva el cumplimiento de unas condiciones previas sobre los parámetros la comprobación de la validez de los argumentos se realizarán fuera del método y antes de la llamada a la función.

Use filtros u otros mecanismos de control antes de llamar a la función.

V.2.4. Llamadas entre métodos dentro del propio objeto

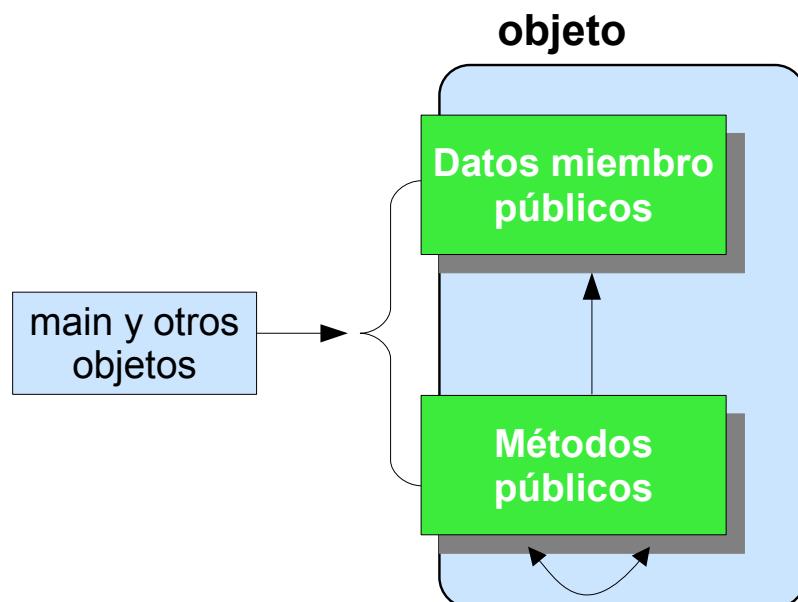
Ya sabemos cómo ejecutar los métodos sobre un objeto:

```
objeto.UnMetodo(...);
```

¿Pero pueden llamarse dentro del objeto unos métodos a otros? Sí.

Todos los métodos de un objeto pueden llamar a los métodos del mismo objeto. La llamada se especifica indicando el *nombre del método*. No hay que anteponer el nombre de ningún objeto.

```
class MiClase{  
public:  
    void UnMetodo(parámetros formales){  
    }  
    void OtroMetodo(...){  
        UnMetodo (parámetros reales);  
    }  
};
```



Ejemplo. Implemente el método Traslada para que traslade el segmento tanto en horizontal como en vertical.

```
class SegmentoDirigido{  
public:  
    double x_1, y_1;    // Valores indeterminados  
    double x_2, y_2;    // Valores indeterminados  
  
    void TrasladaHorizontal (double unidades){  
        x_1 = x_1 + unidades;  
        x_2 = x_2 + unidades;  
    }  
    void TrasladaVertical (double unidades){  
        y_1 = y_1 + unidades;  
        y_2 = y_2 + unidades;  
    }  
  
    // Traslada en Horizontal y en Vertical  
  
    void TrasladaRepetiendoCodigo (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);  
    }  
    ....  
};
```


472

```
int main(){
    SegmentoDirigido un_segmento; // Valores indeterminados

    un_segmento.x_1 = 3.4;
    un_segmento.y_1 = 5.6;
    un_segmento.x_2 = 4.5;
    un_segmento.y_2 = 2.3;

    cout << "Segmento Dirigido." << endl;
    cout << endl;

    cout << "Antes de la traslación:" << endl;
    cout << un_segmento.x_1 << " , " << un_segmento.y_1; // 3.4 , 5.6
    cout << endl;
    cout << un_segmento.x_2 << " , " << un_segmento.y_2; // 4.5 , 2.3
    cout << endl;

    un_segmento.Traslada(5.0, 10.0);

    cout << endl;
    cout << "Después de la traslación:" << endl;
    cout << un_segmento.x_1 << " , " << un_segmento.y_1; // 8.4 , 15.6
    cout << endl;
    cout << un_segmento.x_2 << " , " << un_segmento.y_2; // 9.5 , 12.3
    cout << endl;
}
```

Dentro de la clase, los métodos pueden llamarse unos a otros. Esto nos permite cumplir el principio de una única vez.

Ejercicio. Aplique un interés porcentual al saldo de la cuenta bancaria.

```
class CuentaBancaria{  
public:  
    double saldo;           // Valor indeterminado  
    string identificador;  // ""  
  
    // PRE: tanto_porcentaje > 0  
    void AplicaInteresPorcentualRepitiendoCodigo(int tanto_porcentaje){  
        double cantidad = saldo * tanto_porcentaje / 100.0;  
        saldo = saldo + cantidad;  
    }  
    // PRE: tanto_porcentaje > 0  
  
    void AplicaInteresPorcentual (int tanto_porcentaje){  
        Ingresar (saldo * tanto_porcentaje / 100.0);  
    }  
  
    // PRE: 0 < cantidad  
    void Ingresar(double cantidad){  
        saldo = saldo + cantidad;  
    }  
    // PRE: 0 < cantidad <= saldo  
    void Retira(double cantidad){  
        saldo = saldo - cantidad;  
    }  
};  
int main(){  
    CuentaBancaria cuenta; // "", x  
  
    cuenta.saldo = 0;           // "", 0  
    cuenta.identificador = "20310381450100006529"; // "2...9", 0  
    una_cuenta.Ingresar(25);      // "2...9", 25  
    una_cuenta.AplicaInteresPorcentual(3); // "2...9", 25.75
```



V.3. Ocultación de información

V.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.

Con la cuenta bancaria tenemos dos formas de ingresar 25 euros:

```
cuenta.Ingresa(25);  
cuenta.saldo = cuenta.saldo + 25;
```

¿Y si hubiésemos puesto -3000?

```
int main()  
    CuentaBancaria cuenta; // "", x  
  
    cuenta.identificador = "20310381450100006529"; // "2...9", x  
    cuenta.saldo = 0; // "2...9", 0  
  
    cuenta.Ingresa(-3000); // "2...9", 0   
    cuenta.saldo = -3000; // "2...9", -3000 
```

Para controlar las operaciones válidas sobre el saldo, siempre deberíamos usar el método `Ingresa` 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:

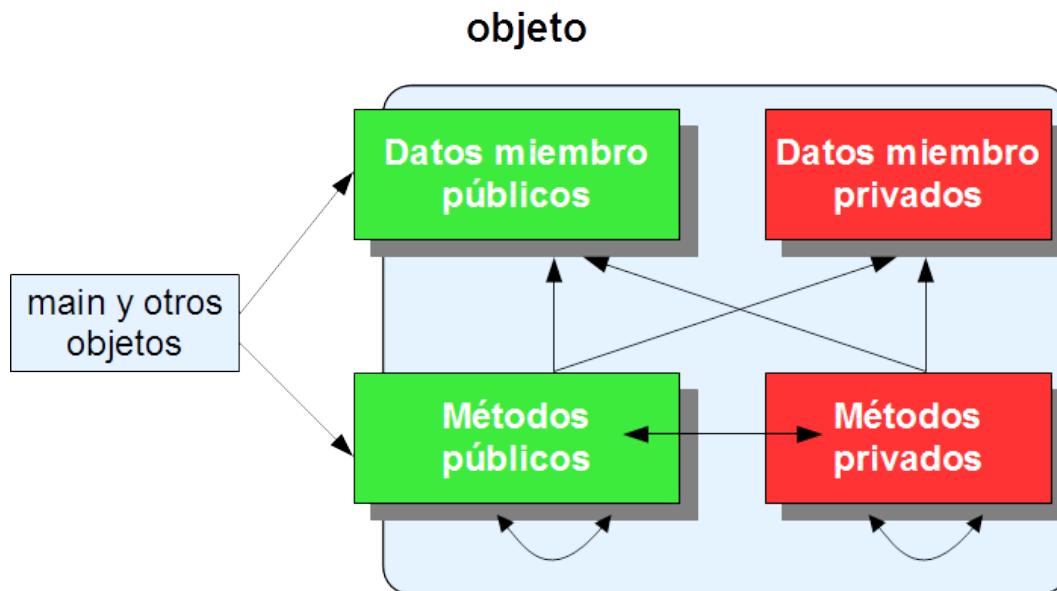
- ▷ **Ámbito público (Public scope)** .
 - Se indica con el especificador de ámbito `public`
 - Los miembros públicos son visibles dentro y fuera del objeto.
- ▷ **Ámbito privado (Private scope)** .
 - Se indica con el especificador de ámbito `private` (éste es el ámbito por defecto si no se pone nada)

```
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.



En la representación gráfica de las clases, los datos miembro privados se notan con signo -. La gráfica de la izquierda sería una representación con el estándar **UML** (Unified Modeling Language) *original*. Nosotros usaremos una adaptación como aparece a la derecha, semejante a la sintaxis de C++.

Clase
- dato_privado: int + dato_publico: double
- MetodoPrivado(param: double): int + MetodoPublico(param: int): void

UML *puro*



Clase
- int dato_privado + double dato_publico
- int MetodoPrivado(double param) + void MetodoPublico(int param)

UML *adaptado*

V.3.2. Datos miembro privados

Supongamos que cambiamos el ámbito público de los datos miembro de la clase `Cuenta_Bancaria` a privado.

```
class CuentaBancaria{  
private:  
    double saldo;  
    string identificador;  
  
public:  
    // PRE: 0 < cantidad  
    void Ingresa(double cantidad){  
        saldo = saldo + cantidad;  
    }  
    // PRE: 0 < cantidad <= saldo  
    void Retira(double cantidad){  
        saldo = saldo - cantidad;  
    }  
    // PRE: tanto_porcentual > 0  
    void AplicaInteresPorcentual(int tanto_porcentual){  
        Ingresa (saldo * tanto_porcentual / 100.0);  
    }  
};  
int main(){  
    CuentaBancaria cuenta;
```

Las siguientes sentencias provocan un *error de compilación* ya que los datos miembro son ahora *privados*:

```
cuenta.identificador = "20310381450100006529";  
cout << cuenta.saldo;
```

¿Cómo accedemos a ellos para modificarlos/consultarlos?

Al trabajar con datos miembro privados debemos añadirle a la clase:

- ▷ **Métodos para modificar los datos miembro (aquellos que se deseen modificar desde fuera)**
- ▷ **Métodos para obtener el valor actual de los datos miembro (aque-lllos a los que se desee acceder desde fuera)**

```
class CuentaBancaria{  
    private:  
        double saldo;  
        string identificador;  
  
    public:  
  
        void SetIdentificador(string identificador_cuenta){  
            identificador = identificador_cuenta;  
        }  
        string GetIdentificador(){  
            return identificador;  
        }  
  
        // PRE: cantidad > 0  
        void SetSaldo(double cantidad){  
            saldo = cantidad;  
        }  
        double GetSaldo(){  
            return saldo;  
        }  
  
        // PRE: 0 < cantidad  
        void Ingresa(double cantidad){  
            saldo = saldo + cantidad;  
        }  
}
```

```
// PRE: 0 < cantidad <= saldo
void Retira(double cantidad){
    saldo = saldo - cantidad;
}

// PRE: tanto_porcentual > 0
void AplicaInteresPorcentual(int tanto_porcentual){
    Ingresa (saldo * tanto_porcentual / 100.0);
}

};

int main(){
    CuentaBancaria cuenta; // "", x

    cuenta.SetIdentificador("20310381450100006529"); // "2...9", x
    cuenta.SetSaldo(0); // "2...9", 0

    cuenta.Ingresa(25); // "2...9", 25
    cout << cuenta.GetSaldo(); // <- Paréntesis, método void
}
```

Estamos obligados a usar el método SetSaldo para inicializar el campo saldo pero la incorporación de este método introduce una fuente de peligro a la integridad de la clase: en cualquier momento podríamos establecer el valor del campo saldo de manera arbitraria, sin ser consecuencia de Ingresa o Retira.

cuenta.SetSaldo(20000);

Parece razonable no incluir un método SetSaldo pues los cambios en el saldo deberían hacerse siempre con Ingresa y Retira.

Bastará eliminar el método SetSaldo, pero necesitamos un mecanismo para inicializar un objeto cuando se crea. Los constructores (sección V.4 proporcionan esta herramienta básica).

Los métodos permiten establecer la política de acceso a los datos miembro, es decir, determinar cuáles son las operaciones permitidas con ellos.

La clase nos quedaría, temporalmente, así:

CuentaBancaria
- double saldo
- string identificador
+ void SetIdentificador (string identificador_cuenta)
+ string GetIdentificador()
+ void SetSaldo(double cantidad)
+ double GetSaldo()
+ void Ingresa(double cantidad)
+ void Retira(double cantidad)
+ void AplicaInteresPorcentual(int tanto_por_ciento)



Salvo casos muy excepcionales ¿los hay? no definiremos datos miembro públicos. Siempre serán privados.

IMPORTANT

El acceso a ellos desde fuera de la clase se hará a través de métodos públicos de la clase.

Interfaz (Interface) de una clase: Es el conjunto de datos y métodos públicos de dicha clase. Como usualmente los datos son privados, el término interfaz suele referirse al conjunto de métodos públicos.

Ejemplo. 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	GetDia()
+ int	GetMes()
+ int	GetAnio()
+ string	ToString()

```
class Fecha {  
private:  
    int dia, mes, anio;  
public:  
    // PRE: el_anio >= 1850 && el_anio <= 3000  
    // PRE: 1 <= el_dia && el_dia <= 31  
    // PRE: 1 <= el_mes && el_mes <= 12  
    void SetDiaMesAnio (int el_dia, int el_mes, int el_anio){  
        dia = el_dia;  
        mes = el_mes;  
        anio = el_anio;  
    }  
    int GetDia(){  
        return dia;  
    }  
    int GetMes(){  
        return mes;  
    }  
    int GetAnio(){  
        return anio;  
    }  
}
```

```
string ToString(){
    return to_string(dia) + "/" +
           to_string(mes) + "/" +
           to_string(anio);
    /* También sería correcto (pero poco práctico):
       return to_string(GetDia()) + "/" +
              to_string(GetMes()) + "/" +
              to_string(GetAnio()); */
}
};

int main(){
    Fecha nacimiento_cliente; // Valores indeterminados

    nacimiento_cliente.SetDiaMesAnio(27, 2, 1967); // 27, 2, 1967
    cout << nacimiento_cliente.ToString();

    Fecha hoy = GetFechaHoy(); // Supongamos que esta función existe

    int edad_cliente = hoy.GetAnio() - nacimiento_cliente.GetAnio();
    if ((hoy.GetMes() < nacimiento_cliente.GetMes()) ||
        ((hoy.GetMes() == nacimiento_cliente.GetMes()) &&
         (hoy.GetDia() < nacimiento_cliente.GetDia())))
        edad_cliente--;

    cout << "Su edad es = " << edad_cliente << endl;
}
```

Nota: La comprobación sobre el día, mes y año debe mejorarse ya que no todos los meses tienen los mismos días y considerar los años bisiestos. Lo haremos posteriormente.

Ejercicio. Cambiemos el ámbito (de public a private) de los datos miembro de la clase SegmentoDirigido. Obligamos cambiar los 4 datos simultáneamente.

SegmentoDirigido
<pre>- 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)</pre>

```
class SegmentoDirigido{

private:
    double x_1, y_1;
    double 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;
    }
}
```

```
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; // Valores indeterminados

un_segmento.SetCoordenadas(3.4, 5.6, 4.5, 2.3);

cout << "Segmento Dirigido." << endl;
cout << "Antes de la traslación:" << endl;
cout << un_segmento.OrigenAbscisa() << " , "
    << un_segmento.OrigenOrdenada();           // 3.4 , 5.6
cout << endl;
cout << un_segmento.FinalAbscisa() << " , "
    << un_segmento.FinalOrdenada();           // 4.5 , 2.3
cout << endl;
```

```
un_segmento.Traslada(5.0, 10.0);

cout << "Después de la traslación:" << endl;
cout << un_segmento.OrigenAbscisa() << " , "
    << un_segmento.OrigenOrdenada();           // 8.4 , 15.6
cout << endl;
cout << un_segmento.FinalAbscisa() << " , "
    << un_segmento.FinalOrdenada();           // 9.5 , 12.3
cout << endl;

SegmentoDirigido otro_segmento;    // Valores indeterminados
otro_segmento.SetCoordenadas (3.4, 5.6, 3.4, 5.6);
....
```

Ejercicio. Representemos una circunferencia.

Circunferencia
- double centro_x - double centro_y - double radio + void SetCentro(double abscisa, double ordenada) + void SetRadio(double el_radio) + double AbscisaCentro() + double OrdenadaCentro() + double Radio() + double Longitud() + double Area() + void Traslada(double en_horizontal, double en_vertical)

```
const double PI = 3.1415927;

class Circunferencia{
private:
    double centro_x, centro_y;
    double radio;      // PRE: radio > 0
public:
    void SetCentro(double abscisa, double ordenada){
        centro_x = abscisa;
        centro_y = ordenada;
    }
    // PRE: el_radio > 0
    void SetRadio(double el_radio){
        radio = el_radio;
    }
    double AbscisaCentro(){
        return centro_x;
    }
    double OrdenadaCentro(){
        return centro_y;
    }
```

```
double Radio(){
    return radio;
}
double Longitud(){
    return 2*PI*radio;
}
double Area(){
    return PI*radio*radio;
}
void Traslada(double en_horizontal, double en_vertical){
    centro_x = centro_x + en_horizontal;
    centro_y = centro_y + en_vertical;
}
};

int main(){
    Circunferencia mi_arro; // Valores indeterminados
    double longitud_mi_arro;

    mi_arro.SetCentro(4.5, 6.7);
    mi_arro.SetRadio(2.1);
    longitud_mi_arro = mi_arro.Longitud(); // 13.19468

    mi_arro.SetRadio(8.3); // Cambiamos el radio
    longitud_mi_arro = mi_arro.Longitud(); // 52,15044

    mi_arro.Traslada(10.1, 15.2); // Traslación
    longitud_mi_arro = mi_arro.Longitud(); // 52,15044
    .....
```

V.3.3. Métodos privados

Si tenemos que realizar un mismo conjunto de operaciones en varios sitios de la clase, usaremos un método para así no repetir código. Si no queremos que se pueda usar desde fuera de la clase, lo declaramos `private`.

Los métodos private se reservan para tareas internas de la clase que no son necesarias (ni convenientes) ofrecer a los clientes de la clase.

Ejemplo. Sobre la clase `CuentaBancaria`, supongamos que todas las cuentas bancarias empiezan por `ES0023`. Podríamos evitar que el usuario escriba este valor cuando se introduce un número de cuenta.

```
class CuentaBancaria{  
    private:  
        double saldo;  
        string identificador;  
  
        string PrefijoCuenta (void){  
            return "ES0023";  
        }  
    public:  
        void SetIdentificador(string identificador_cuenta){  
            identificador = PrefijoCuenta()+identificador_cuenta;  
        }  
        string GetIdentificador(){  
            return identificador;  
        }  
        .....  
};
```

```
int main(){
    CuentaBancaria cuenta;    // "", x

    cuenta.SetIdentificador("6529"); // "ES00236529", x
    cuenta.SetSaldo(0);           // "ES00236529", 0
    .....
```

Ejercicio. Añadimos el método Normaliza a la clase Fraccion. Para ello, implementamos el cómputo del MCD con un método privado.

Fraccion	
- int	numerador
- int	denominador
- int	MCD()
+ void	SetNumerador (int el Numerador)
+ void	SetDenominador (int el denominador)
+ void	SetTerminos (int el Numerador, int el denominador)
+ int	Numerador()
+ int	Denominador()
+ double	Division()
+ double	Normaliza()

```
class Fraccion{

    private:
        int numerador;
        int denominador;

        int MCD (){
            // Cálculo del MCD entre numerador y denominador.
            // Es un método de la clase, por lo que puede usar los
            // campos numerador y denominador sin necesidad de que sean
            // parámetros formales.

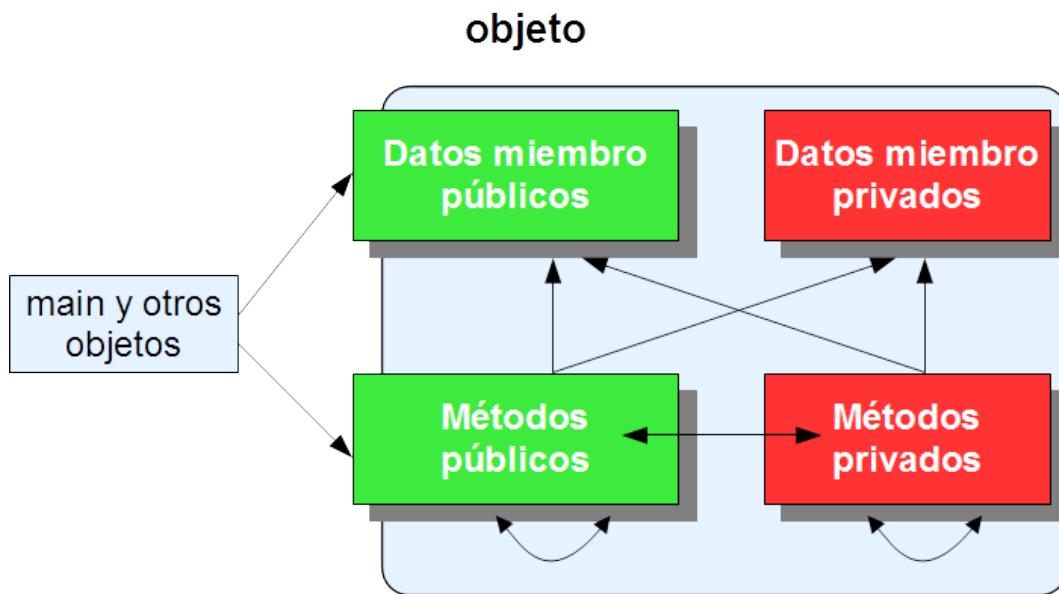
            .....
    }

    public:
        void SetNumerador (int el Numerador){
            numerador = el Numerador;
        }

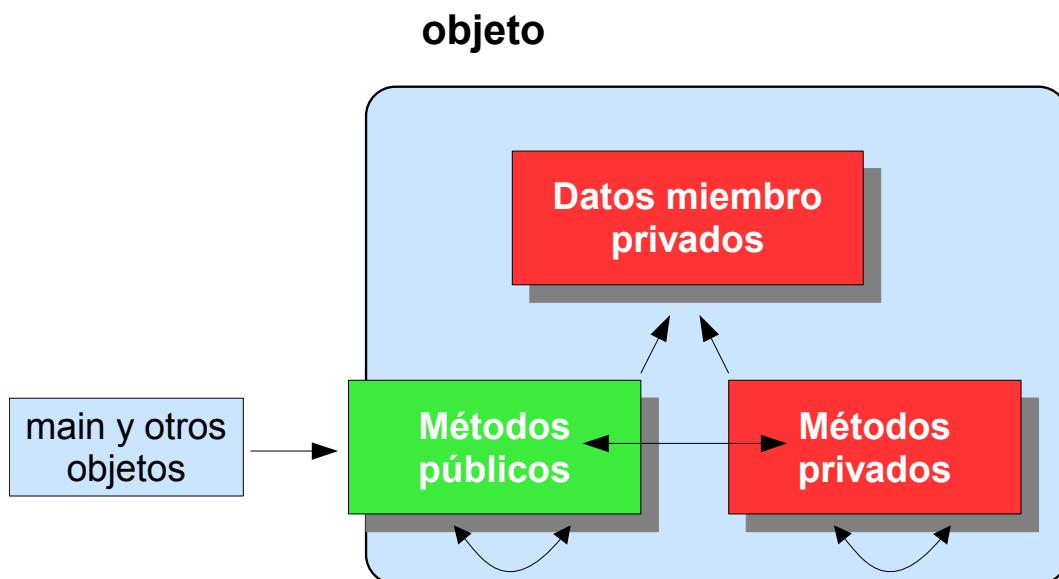
        void SetDenominador (int el denominador){
            denominador = el denominador;
        }
}
```

```
void SetTerminos (int el_numerator, int el_denominador){  
    SetNumerador (el_numerator);  
    SetDenominador (el_denominador);  
}  
  
int Numerador(){  
    return numerador;  
}  
int Denominador(){  
    return denominador;  
}  
double Division(){  
    return 1.0 * numerador / denominador;  
}  
  
void Normaliza(){  
    int mcd = MCD(); // Se llama al método privado  
  
    numerador = numerador/mcd;  
    denominador = denominador/mcd;  
}  
};  
  
int main(){  
  
    Fraccion una_fraccion; // Valor indeterminado  
  
    una_fraccion.SetTerminos (4, 10); // 4/10  
    division_real = una_fraccion.Division(); // 0.4  
    una_fraccion.Normaliza(); // 2/5  
  
    double division_real = una_fraccion.Division(); // 0.4  
}
```

En resumen. Lo que C++ permite:



Lo que nosotros haremos:



V.4. Constructores

V.4.1. Estado inválido de un objeto

¿Qué ocurre si una variable no tiene asignado un valor? Cualquier operación que realicemos con ella devolverá un valor *indeterminado*.

```
int main(){
    int euros_enteros;      // Valor indeterminado (basura)
    cout << euros_enteros; // Imprime un valor indeterminado
```

Lo mismo ocurre con un objeto que contenga datos miembro sin un valor determinado.

```
class Clase{
private:
    int dato; // Valor indeterminado
public:
    void SetDato(int valor){
        dato = valor;
    }
    int GetDato(){
        return dato;
    }
};

int main(){
    Clase objeto;           // Valor indeterminado
    cout << objeto.GetDato(); // Imprime un valor indeterminado
                           // Estado inválido
```

Diremos que un objeto se encuentra en un momento dado en un *estado inválido* si algunos de sus datos miembros *esenciales* no tienen un valor correcto.

Queremos obligar a 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.

El uso de *valores por defecto* puede ser útil en las siguientes situaciones:

- ▷ Una circunferencia por defecto podría ser la circunferencia goniométrica (centrada en el origen y de radio 1)
- ▷ Una cuenta bancaria se crea por defecto con saldo 0.
- ▷ Un segmento dirigido por defecto es el que va del origen (0, 0) al mismo origen (0, 0) (segmento degenerado)

Sin embargo, los valores por defecto no resuelven otros problemas:

- ▷ ¿Qué identificador por defecto asociamos a una nueva cuenta bancaria? No nos vale un valor nulo (). Debe ser un identificador válido.
- ▷ Si no admitimos segmentos degenerados, ¿cuál podría ser un segmento por defecto? ¿(0, 0)-(0, 1)? ¿(0, 0)-(1, 0)?,
- ▷ Si tenemos la clase Persona, ¿qué nombre, DNI, edad, etc. se le asignan por defecto? No tiene sentido.

Vamos a ver una herramienta más potente: el constructor. Éste nos permitirá, entre otras cosas, dar valores concretos a los datos miembro en el momento de la definición del objeto. Esos valores serán los valores iniciales del objeto cuando se cree (en su declaración).

Para evitar que un objeto esté en un estado inválido:

- ▷ *Cuando tenga sentido, podemos asignar valores por defecto a los datos miembro.*
- ▷ *En cualquier caso obligaremos a que en el momento de la creación de cada objeto se le suministren unos valores concretos y correctos. Para ello, se recurre a los constructores.*

V.4.2. Definición de constructores

Un **constructor (constructor)** es *como* un método sin tipo de la clase (lo que significa que NO devuelve nada). En él se incluyen todas las acciones que queramos realizar en el momento de construir un objeto, cuando empieza su existencia.

- ▷ El constructor se define dentro de la clase, en la sección `public`.
- ▷ Debe llamarse obligatoriamente, igual que la clase. No se pone `void`, sino únicamente el nombre de la clase.
- ▷ Cuando se crea un objeto, el compilador ejecutará las instrucciones especificadas en el constructor.
- ▷ Se pueden incluir parámetros.

Para asignar 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)**

Su sintaxis es: `<dato_miembro> (<valor_inicial>)` por cada campo, y se forma una lista de iniciación separándolas por comas.

La lista de inicialización del constructor se escribe tras los parámetros formales, precedida de los dos puntos (`:`)

Los constructores nos permiten ejecutar automáticamente un conjunto de instrucciones, cada vez que se crea un objeto.

En particular, si se les pasan valores adecuados a los parámetros formales, éstos permiten la creación de un objeto en un estado válido en el mismo momento de su creación.

Ejemplo. Añada un constructor a la cuenta bancaria pasándole como parámetro el identificador de cuenta.

```
class CuentaBancaria{  
private:  
    double saldo;  
    string identificador;  
    .....  
public:  
    /*  
     * CuentaBancaria (string identificador_cuenta) : saldo(0) { // Correcto  
     *     identificador = identificador_cuenta;  
     }  
     */  
  
    // Preferimos esta manera de escribir el constructor  
    CuentaBancaria (string identificador_cuenta)  
        : saldo(0), // <- Inicializador de saldo  
        identificador(identificador_cuenta) // <- Inic. de identificador  
    {}  
    .....  
};  
int main(){  
    // CuentaBancaria cuenta; <- Error de compilación  
  
    CuentaBancaria cuenta("20310381450100007510");  
    // cuenta: "20310381450100007510", 0.0  
    // Estado válido
```

Si los parámetros actuales son variables habrá que crear el objeto después de establecer dichas variables.

```
cin >> identificador;  
CuentaBancaria cuenta(identificador);
```

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
- string identificador
+ CuentaBancaria (string identificador_cuenta)
- void SetIdentificador (string identificador_cuenta)
+ string GetIdentificador()
+ 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");
```

Ejercicio. Obligamos a pasar los cuatro puntos de un SegmentoDirigido en el constructor. No queremos un segmento por defecto.

SegmentoDirigido
- double x_1
- double y_1
- double x_2
- double y_2
+ SegmentoDirigido(double origen_abscisa, double origen_ordenada, double final_abscisa, double final_ordenada)
.....

```
class SegmentoDirigido {  
private:  
    double x_1, y_1;  
    double x_2, y_2;  
public:  
    SegmentoDirigido(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; <- Error de compilación  
  
    SegmentoDirigido un_segmento(3.4, 5.6, 4.5, 2.3); // estado válido  
    .....  
}
```

Ejercicio. Añadimos un constructor a la clase Fecha 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,    // PRE: 1 <= dia <= 31  
    mes,      // PRE: 1 <= mes <= 12  
    anio;     // PRE: anio >= 1850 && el_anio <= 3000  
    .....  
public:  
  
    // PRE: 1 <= dia <= 31  
    // PRE: 1 <= mes <= 12  
    // PRE: anio >= 1850 && el_anio <= 3000  
    Fecha(int el_dia, int el_mes, int el_anio)  
        : dia (el_dia), mes (el_mes), anio (el_anio)  
    {}  
    .....  
};  
int main(){  
    // Fecha nacimiento_cliente; // Error de compilación  
  
    Fecha nacimiento_cliente (27, 2, 1967);      // {27, 2, 1967}  
    .....  
}
```

Dentro de un constructor podemos ejecutar código y llamar a métodos de la clase.

CuentaBancaria
- double saldo
- string identificador
+ CuentaBancaria (string identificador_cuenta)
- string PrefijoCuenta (void)
- void SetIdentificador (string identificador_cuenta)
+ string GetIdentificador()
+ double Saldo()
+ void Ingresa(double cantidad)
+ void Retira(double cantidad)
+ void AplicaInteresPorcentual(int tanto_porcentual)

```
class CuentaBancaria{  
private:  
    double saldo;  
    string identificador;  
  
    string PrefijoCuenta (void){  
        return "ES0023";  
    }  
    void SetIdentificador (string identificador_cuenta) {  
        identificador = PrefijoCuenta()+identificador_cuenta;  
    }  
public:  
    CuentaBancaria (string identificador_cuenta) : saldo(0)  
    {  
        SetIdentificador (identificador_cuenta);  
    }  
    string GetIdentificador() {  
        return identificador;  
    }
```

```
int main(){

    CuentaBancaria cuenta("6529");      // {"ES00236529", 0}

    // Las siguientes sentencias darían un error de compilación
    // por utilizar métodos privados de "CuentaBancaria"
    // desde la función main

    // cuenta.SetIdentificador("ES00231234");
    // cuenta.SetIdentificador(cuenta.PrefijoCuenta()+"1234");
}
```

V.4.3. Constructores sin parámetros

Si se desea, se puede proporcionar un constructor sin parámetros. Hay que indicar que, internamente, C++ proporciona un constructor de oficio sin parámetros -oculto- que permite crear el objeto y poco más.

```
class MiClaseSinConstructor {  
    .....  
};  
int main(){  
    MiClaseSinConstructor objeto; // Constructor oculto de C++  
}
```

Si el programador define cualquier constructor (con o sin parámetros) ya no está disponible el constructor de oficio.

Si lo necesitara, deberá implementarlo.

En el siguiente ejemplo ya no está disponible el constructor sin argumentos una vez que se ha definido otro constructor.

```
class MiClase{  
    .....  
public:  
    MiClase (int un_parametro) { // Constructor con argumentos  
        .....  
    }  
    .....  
};  
  
int main(){  
    MiClase objeto; // Error de compilación  
    .....  
}
```

Si lo necesitamos hay que implementarlo (aunque no haga nada).

```
class MiClase{  
    ....  
public:  
  
    MiClase() { // Constructor sin argumentos  
        ....  
    }  
  
    MiClase (int un_parametro) { // Constructor con argumentos  
        ....  
    }  
    ....  
};  
  
int main(){  
    MiClase objeto1;      // Actúa el constructor sin argumentos  
    MiClase objeto2 (5); // Actúa el constructor con argumentos  
    ....
```

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)**

Ejemplo. Sobre la clase Fecha, añadimos un constructor sin parámetros para que cree una fecha con los datos de la fecha actual.

Fecha
- int dia
- int mes
- int anio
+ Fecha ()
+ Fecha (int el_dia, int el_mes, int el_anio)
... ...

El código del constructor sin argumentos de la clase Fecha podría ser el siguiente. Debemos usar la biblioteca `ctime`. No hay que comprender el código del constructor, sino entender cuándo se ejecuta.

```
Fecha() {
    time_t momento;

    time (&momento); // Num. segundos desde 00:00, 1 Ene 1970 UTC

    // Puntero a un "struct tm" con fecha/hora local
    tm * ahora = localtime(&momento);

    // Escribir en los campos desde los valores del struct "ahora"
    anio = 1900 + ahora->tm_year;
    mes = ahora->tm_mon+1;
    dia = ahora->tm_mday;
}

int main(){
    Fecha fecha_hoy;          // Constructor sin parámetros

    cout << "Hoy es " << fecha_hoy.ToString() << endl;
```

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 que *parecen* aleatorias.

Vamos a crear una clase para generar números 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.

```
#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_int_distribution<int> 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;

    for (int i=0; i<100; i++)
        cout << aleatorio.Siguiente() << " ";
}
```

V.4.4. Sobrecarga de constructores

Se puede proporcionar más de un constructor, siempre que cambien en el tipo o en el número de parámetros. El compilador creará el objeto llamando al constructor correspondiente según corresponda con los parámetros actuales.

Ejemplo. Queremos dos constructores para la clase Fecha:

Fecha
- int dia
- int mes
- int anio
+ Fecha()
+ Fecha(int el_dia, int el_mes, int el_anio)
.....

```
class Fecha{  
private:  
    int dia, mes, anio;  
    .....  
public:  
    Fecha() {  
        .....  
    }  
    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 un_dia (dia, mes, anio); // Constructor con parámetros;  
    .....  
}
```

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 primer ingreso (primer saldo).

CuentaBancariaInternacional
- double saldo
- double identificador
+ CuentaBancariaInternacional(string identificador_cuenta)
+ CuentaBancariaInternacional(string identificador_cuenta, double saldo_inicial)
.....

- double saldo
- double identificador
+ CuentaBancariaInternacional(string identificador_cuenta)
+ CuentaBancariaInternacional(string identificador_cuenta, double saldo_inicial)
.....

```
class CuentaBancariaInternacional {  
private:  
    double saldo;  
    string identificador;  
    .....  
public:  
    CuentaBancariaInternacional(string identificador_cuenta){ // 1  
        : identificador(identificador_cuenta) {}  
  
    CuentaBancariaInternacional(string identificador_cuenta,  
                                double saldo_inicial){ // 2  
        : saldo (saldo_inicial), identificador(identificador_cuenta) {}  
    .....  
};  
int main(){  
    CuentaBancariaInternacional cuenta1("IT465775-7511"); // 1  
    // cuenta1: "IT465775-7511", 0.0  
    CuentaBancariaInternacional cuenta2("BE3456-776-995-994", 3000); // 2  
    // cuenta2: "BE3456-776-995-994", 3000
```

V.4.5. Llamadas entre constructores

Si observamos el ejemplo anterior, podemos apreciar que hay cierto código repetido en los constructores.

```
    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 (clase CuentaBancariaInternacional).

```
class CuentaBancariaInternacional {  
private:  
    double saldo;  
    string identificador;  
    .....  
public:  
    CuentaBancariaInternacional(string identificador_cuenta){ // 1  
        : identificador(identificador_cuenta) {}  
  
    CuentaBancariaInternacional (string identificador_cuenta,  
                                double saldo_inicial){ // 2  
        : CuentaBancariaInternacional (identificador_cuenta),  
        saldo (saldo_inicial) {}  
    .....  
};  
  
int main(){  
    CuentaBancariaInternacional cuenta1("IT465775-7511"); // 1  
    // cuenta1: "IT465775-7511", 0.0  
    CuentaBancariaInternacional cuenta2("BE3456-776-995-994", 3000); // 2  
    // cuenta2: "BE3456-776-995-994", 3000
```

V.4.6. Estado inválido de un objeto -revisión-

¿Qué ocurre si los datos suministrados en el constructor no son correctos? Tenemos un problema cuya resolución requiere de herramientas que no se verán en este curso.

Ejemplo. Retomamos el ejemplo de la clase Fecha.

El constructor asignaba los parámetros actuales a los datos miembro:

```
public:  
    Fecha(int el_dia, int el_mes, int el_anio)  
        : dia(el_dia), mes(el_mes), anio(el_anio)
```

Pero, ¿qué ocurre si los parámetros actuales no son correctos?

```
int main(){  
    Fecha una_fecha(-1, 2, -9); // -1, 2 ,-9  
    // una_fecha está en un estado inválido
```

Una solución a este problema pasa por *lanzar una excepción* dentro del constructor y no permitir la creación del objeto, pero está fuera de los objetivos del curso. Al menos, podríamos paliar el problema asignando un valor concreto a los datos miembro que indique el problema.

- ▷ 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 no le podemos asignar ningún valor especial ya que cualquier pareja de reales puede formar un centro.
- ▷ A un objeto de la clase CuentaBancaria, le asignaríamos NAN al saldo y el valor nulo al identificador.
- ▷ A un objeto de la clase SegmentoDirigido, le asignaríamos NAN a las 4 coordenadas.

En resumen:

Para evitar que un objeto esté en un estado inválido en el mismo momento de su definición:

Si los valores de los parámetros pasados al constructor no son correctos, la mejor solución pasaría por lanzar una excepción (se verá en un tema posterior de ampliación) desde dentro del constructor y así impedir la creación del objeto.

La "solución"que adoptaremos será el **uso intensivo de precondiciones**.

- ▷ Se indican tanto en los datos como en los métodos.
- ▷ Se cede la responsabilidad al programador usuario de la clase.

El problema no se resuelve sin más: si el programador no es cuidadoso y empieza a actuar el constructor, no hay nada que hacer.

El programador debe comprobar, antes de que actúe el constructor, la validez de los datos que le suministrará de acuerdo a las precondiciones.

```
class Fecha {  
  
private:  
    int dia;      // PRE: 1<=dia<=31  
    int mes;      // PRE: 1<=mes<=12  
    .....  
  
public:  
    .....  
    // PRE: 1<=el_dia<=31  
    // PRE: 1<=el_mes<=12  
    .....  
    Fecha (int el_dia, int el_mes, int el_anio) {  
        .....  
    }  
};  
  
int main () {  
  
    int dia_hoy, mes_hoy, anio_hoy;  
  
    do {      // Leer y filtrar dia_hoy  
        cin >> dia_hoy;  
    } while (dia_hoy<1 || dia_hoy>31);  
  
    do {      // Leer y filtrar mes_hoy  
        cin >> mes_hoy;  
    } while (mes_hoy<1 || mes_hoy>12);  
    .....  
  
    // Se cumplen la precondiciones --> Creo un objeto  
    Fecha hoy (dia_hoy, mes_hoy, anio_hoy);
```

V.5. Copiando objetos

V.5.1. Operador de asignación

C++ permite asignar objetos entre sí a través del *operador de asignación por defecto (default assignment operator)* =

Este operador lo define automáticamente el compilador para cualquier clase, y funciona como si trabajásemos con tipos básicos:

```
objeto = otro_objeto;
```

- ▷ Ambos objetos han de ser de la misma clase.
- ▷ Ambos objetos se han creado previamente, existen (en este momento parece evidente e innecesaria esta afirmación).
- ▷ Se realiza una copia de **todos** los datos miembro, **tanto los privados como los públicos**.
- ▷ En realidad se realiza una copia literal, bit a bit, desde otro_objeto a objeto. El resultado es que objeto es un **clon** de otro_objeto.
- ▷ Se dice *por defecto* porque puede usarse sin más. Está disponible para la asignación entre objetos de cualquier clase que definamos (entiéndase: entre objetos de la misma clase, para todas las clases).
- ▷ No tiene sentido hablar de *copiar* los métodos: ¡ya estaban disponibles!

El operador de asignación por defecto permite asignar objetos entre sí. Se copia el estado del objeto.

```
class MiClase{  
private:  
    int privada;  
public:  
    int publica;  
  
    MiClase(int priv, int pub) : privada(priv), publica(pub)  
    { }  
  
    int Privada() {  
        return privada;  
    }  
  
    int Publica() {  
        return publica;  
    }  
};  
  
int main(){  
  
    MiClase objeto(1,3);          // {privada=1, publica=3}  
    MiClase otro_objeto(6,8);     // {privada=6, publica=8}  
  
    objeto = otro_objeto; // Asignación objeto <- otro_objeto  
  
    cout << "\nPrivada de objeto: " << objeto.Privada(); // 6  
    cout << "\nPública de objeto: " << objeto.Publica(); // 8  
    cout << "\nPrivada de otro_objeto: " << otro_objeto.Privada(); // 6  
    cout << "\nPública de otro_objeto: " << otro_objeto.Publica(); // 8  
}
```

Nota. Recordemos que jamás usaremos datos miembro públicos. El anterior es un ejemplo para mostrar los efectos del operador de asignación.

V.5.2. El constructor de copia

C++ permite *inicializar un objeto que se está creando con los datos de otro*, consiguiendo que el objeto creado sea un clon del que le sirve de modelo.

- ▷ Esta tarea la realiza un *constructor* (*constructor de copia (copy constructor)*) que recibe como parámetro un objeto de la misma clase. El constructor de copia es invocado en el momento de la declaración del objeto, justo en el momento en el que se va a crear.
- ▷ El constructor de copia es proporcionado automáticamente (de oficio) por el compilador.

El constructor de copia permite inicializar el estado de un objeto con los datos de otro objeto, en el momento de su definición, por lo que se copia el estado del objeto.

Por ejemplo, considere la clase Fecha. **El constructor con argumentos crea el objeto fecha_nacimiento:**

```
Fecha fecha_nacimiento (16, 4, 1999); // Const. con argumentos
```

Si queremos otro objeto con una copia literal de fecha_nacimiento bastará con escribir:

```
Fecha copia_fecha_nacimiento (fecha_nacimiento); // Const. copia
```

El objeto copia_fecha_nacimiento se crea (*constructor*) y se inicializa copiando (*constructor de copia*) los valores de fecha_nacimiento.

No hay que implementar nada para que esta copia se realice: el compilador llama automáticamente al constructor de copia, que siempre está disponible.

El constructor de copia puede llamarse con una sintaxis diferente, usando el carácter `=`, lo que confunde a los programadores noveles con el operador de asignación.

Creamos tres objetos de clase Fecha:

```
Fecha nac_Marina (2, 2, 1995); // Const. con argumentos  
Fecha nac_Sergio (16, 4, 1999); // Const. con argumentos  
Fecha dia_importante; // Const. sin argumentos
```

```
Fecha nac_mayor (nac_Marina);
```

← Constructor de copia

```
Fecha nac_menor = nac_Sergio;
```

← Constructor de copia

```
...
```

```
dia_importante = nac_Marina;
```

← Operador de asignación

```
dia_importante = nac_Sergio;
```

En la sentencias

▷ Fecha nac_mayor (nac_Marina);

el compilador llama al *constructor* (está creando el objeto nac_mayor) *de copia* (toma los datos de nac_Marina).

▷ Fecha nac_menor = nac_Sergio;

el compilador *también* llama al *constructor* (está creando el objeto nac_menor) *de copia* (toma los datos de nac_Sergio).

En cambio, en las sentencias

```
dia_importante = nac_Marina;
```

```
dia_importante = nac_Sergio;
```

el compilador llama al operador de asignación (el objeto que actúa como *lvalue*, dia_importante) y los que actúan como *rvalue* (nac_Marina ó nac_Sergio ya existen).

En resumen:

- ▷ **El constructor de copia es invocado en el momento de la declaración del objeto, justo en el momento en el que se va a crear.**

Veremos más adelante que el compilador también invoca automáticamente al constructor de copia cuando se pasa un objeto a una función (*el parámetro formal es un objeto*) y cuando una función devuelve un objeto.

- ▷ C++ proporciona ***dos constructores de oficio***:

- Constructor sin parámetros.
- Constructor de copia.

- ▷ El programador puede definir:

- Constructor(es) con parámetro(s) y/o *su propio, particularizado, constructor sin parámetros*.

Si define algún constructor con parámetros ya no está disponible el constructor sin parámetros de oficio proporcionado por el compilador y tendrá que escribirlo si lo necesita.

- *Su propio, particularizado, constructor de copia*.

En este caso tampoco estará disponible el constructor sin parámetros de oficio proporcionado por el compilador y tendrá que escribirlo si lo necesita.

Nota: Se estudiará en el segundo cuatrimestre.

V.6. 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.

```
class CuentaBancaria {  
  
private:  
    double saldo;  
    string identificador;  
  
    string PrefijoCuenta (void){  
        return "ES0023";  
    }  
    void SetIdentificador(string identificador_cuenta){  
        identificador = PrefijoCuenta()+identificador_cuenta;  
    }  
  
public:  
    Cuenta (string identificador_cuenta) : saldo(0) {  
        SetIdentificador(identificador_cuenta)  
    }  
  
    Cuenta (string identificador_cuenta, int el_saldo) : saldo(el_saldo){  
        SetIdentificador(identificador_cuenta)  
    }  
    string GetIdentificador(){  
        return identificador;  
    }  
    .....  
};
```

```
int main(){
    CuentaBancaria cuenta("6529"); // {"ES00236529", 0}

    // La siguiente sentencia da error de compilación
    // El método SetIdentificador es private:

    cuenta.SetIdentificador("ES00237511");
}
```

Desde fuera (a la función `main` le parece así) hemos conseguido que el identificador sea *constante*. Realmente no lo es para la clase, nada dice que sea así, cualquier método podría modificarlo.

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.

V.6.1. 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.**
- ▷ **La inicialización debe hacerse con un *constructor*.**
 - **El *valor* se recibe como un *parámetro* más en el constructor.**
 - **La asignación se realiza *obligatoriamente* en la *lista de inicialización del constructor*.**

```
class MiClase{  
private:  
    const double CTE_REAL;  
    const string CTE_STRING;  
public:  
    MiClase (double un_real, string un_string) // Constructor  
        : CTE_REAL (un_real),           // Aquí se le asigna el valor  
          CTE_STRING (un_string) // Aquí se le asigna el valor  
    {  
        CTE_REAL = un_real;           // Error de compilación. No es el sitio  
        CTE_STRING = un_string;      // 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 otro_objeto(5.8, "No"); // CTE_REAL = 5.8, CTE_STRING = "No"
```

Recuerde que en la lista de inicialización se puede dar un valores a todos los datos miembros, no sólo las constantes a nivel de objeto.

De hecho, esta es la forma *recomendada* en C++ de inicializar los datos miembro.

```
class Fecha {  
  
private:  
    int dia;      // PRE: 1<=dia<=31  
    int mes;      // PRE: 1<=mes<=12  
    ....  
  
public:  
    ....  
    // PRE: 1<=el_dia<=31  
    // PRE: 1<=el_mes<=12  
    ....  
    Fecha (int el_dia, int el_mes, int el_anio)  
        : dia (el_dia), mes (el_mes), anio (el_anio)  
    { }  
    ....  
};  
};
```

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)  
        : saldo (saldo_inicial), IDENTIFICADOR (identificador_cuenta)  
    {}  
    CuentaBancaria(string identificador_cuenta)  
        : saldo (0), IDENTIFICADOR (identificador_cuenta)  
    {}  
  
    string GetIdentificador(){  
        return IDENTIFICADOR;  
    }  
    .....  
};  
  
int main(){  
  
    CuentaBancaria una_cuenta("20310087370100001345", 100);  
    .....  
}
```

V.6.2. Constantes a nivel de clase

Todos los objetos de una clase comparten el mismo valor de constante.

El valor a asignar se indica en la inicialización del dato miembro.

En UML, las constantes estáticas se resaltan subrayándolas.

Constantes estáticas NO enteras

- ▷ Se **declaran dentro** de la definición de la clase:

```
class <nombre_clase>{  
    ....  
    static const <tipo> <nombre_cte>;  
    ....  
};
```

- ▷ Se **definen fuera** de la definición de la clase:

```
const <tipo> <nombre_clase>::<nombre_cte> = <valor>;
```

```
class MiClase{  
private:  
    static const double CTE_REAL_PRIVADA;  
    ....  
};  
const double MiClase::CTE_REAL_PRIVADA = 6.7;  
  
int main(){  
    MiClase un_objeto; // CTE_REAL_PRIVADA = 6.7;  
    MiClase otro_objeto; // CTE_REAL_PRIVADA = 6.7;
```

El operador `::` 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)*.

Nota: Se verá con más detalle en el segundo cuatrimestre.

Constantes estáticas enteras

▷ O bien como antes:

```
class MiClase{  
private:  
    static const int CTE_ENTERA_PRIVADA;  
    .....  
};  
const int MiClase::CTE_ENTERA_PRIVADA = 6;
```

▷ Preferiblemente se definen en el mismo sitio de la declaración:

```
class MiClase{  
private:  
    static const int CTE_ENTERA_PRIVADA = 4;  
    .....  
};
```

Vamos a utilizar con mucha frecuencia constantes estáticas enteras para dimensionar vectores dentro de una clase.

V.6.3. El operador de asignación y el constructor de copia en clases con datos miembro constantes

¿Qué ocurre cuando una clase contiene datos miembro constantes?

- ▷ Una **constante estática** es la misma para todos los objetos de una clase. Por tanto, éstas no dan problemas en la asignación entre objetos y menos aún en el constructor de copia.
Una constante estática no se guarda dentro del objeto, sino que es **externa** y todos los objetos de la clase acceden a ella.

```
class SecuenciaCaracteres {  
private:  
    static const int DIM = 50;  
    char vector_privado[DIM];  
    int total_utilizados;  
public:  
    SecuenciaCaracteres() : total_utilizados(0) {}  
    void Aniade (char c) {  
        .....  
    }  
    .....  
};  
int main(){  
    SecuenciaCaracteres secuencia, otra_secuencia;  
    secuencia.Aniade('t');  
    secuencia.Aniade('e');  
    secuencia.Aniade('c');  
    secuencia.Aniade('a');  
    otra_secuencia = secuencia; // Correcto  
    SecuenciaCaracteres otra_secuencia_mas (secuencia); // Correcto
```

- ▷ Una **constante no estática** es propia de cada objeto.

¿Qué sentido tiene cambiar su valor con la constante de otro objeto? Ninguno.

C++ NO permite asignar objetos entre sí (usando el operador de asignación por defecto) cuando la clase contiene datos miembro constantes no estáticos

Si un objeto tiene datos constantes, **sí** podemos inicializar otro objeto con sus datos con un **constructor**. Es fácil de entender: el objeto que se está creando todavía no tenía definida la constante, usa el valor del objeto que sirve de modelo para inicializar su constante.

```
class CuentaBancaria{  
private:  
    const string identificador;  
    .....  
public:  
    CuentaBancaria(string identificador_cuenta, double saldo_inicial)  
        : identificador(identificador_cuenta), saldo(saldo_inicial),  
    {}  
    .....  
};  
int main(){  
    CuentaBancaria cuenta ("Un identificador", 20000);  
    CuentaBancaria otra_cuenta ("Otro identificador", 30000);  
  
    otra_cuenta = cuenta; // Error: datos miembro constantes  
  
    CuentaBancaria otra_cuenta_mas (cuenta); // Correcto
```

V.7. Programando como profesionales

Conceptos fundamentales vistos hasta ahora en la construcción de una clase:

- ▷ **Ámbitos private y public.**
- ▷ **Encapsulación de datos y métodos dentro de una clase.**

Ahora bien,

- ▷ **¿Qué incluimos como dato miembro de una clase y qué pasamos como parámetros a sus métodos?**
- ▷ **¿Qué preferimos: pocas clases que hagan muchas cosas distintas o más clases que hagan cosas concretas?**

V.7.1. Datos miembro vs parámetros de los métodos

Los datos miembro se comportan como datos *globales* dentro del objeto, por lo que son directamente accesibles desde cualquier método definido dentro de la clase y no hay que pasarlos como parámetros a dichos métodos.

Por tanto, ¿los métodos de las clases en PDO suelen tener menos parámetros que las funciones *globales* usadas en Programación Procedural?

¡Sí, por supuesto!

Ejemplo. En la clase SegmentoDirigido, ¿incluimos la longitud de un segmento como dato miembro, o bien la calculamos cuando sea preciso con una función externa o con un método?

- ▷ No debemos incluir longitud como un dato miembro. Es una propiedad que puede calcularse en función de las coordenadas.
- ▷ Con una función debemos pasar cuatro parámetros:

```
Longitud(double x_1, double y_1, double x_2, double y_2){  
    ....  
}
```

- ▷ Dentro de la clase, no:

SegmentoDirigido
- double x_1
- double y_1
- double x_2
- double y_2
.....
+ double Longitud()

Las propiedades que se puedan calcular a partir de los datos miembro, se obtendrán a través de un método.

Ejemplo. Clase Persona. Representaremos su nombre, altura y edad.

Persona	
- string	nombre
- int	edad
- int	altura
- bool	EsCorrectaEdad(string una_edad)
- bool	EsCorrectaAltura(string una_altura)
+	Persona(string nombre_persona, int edad_persona, int altura_persona)
+ void	SetNombre(string nombre_persona)
+ void	SetEdad(int edad_persona)
+ void	SetAltura(int altura_persona)
+ string	Nombre()
+ int	Edad()
+ int	Altura()

```
class Persona{  
private:  
    string nombre;  
    int edad;  
    int altura;  
  
    bool EsCorrectaEdad(int una_edad){  
        return (una_edad > 0 && una_edad < 120);  
    }  
    bool EsCorrectaAltura(int una_altura){  
        return (una_altura > 50 && una_altura < 250);  
    }  
public:  
    Persona(string nombre_persona, int edad_persona, int altura_persona){  
        if (EsCorrectaEdad(edad_persona)  
            && EsCorrectaAltura(altura_persona)) {  
            nombre = nombre_persona;  
            edad = edad_persona;  
            altura = altura_persona;  
        }  
    }  
}
```

```
    }

    else { // Lo mejor sería lanzar una excepción
        nombre = "";
        edad = -1;
        altura = -1;
    }

}

string Nombre(){
    return nombre;
}

int Edad(){
    .....
}

int Altura(){
    .....
}

void SetNombre(string nombre_persona){
    nombre = nombre_persona;
}

void SetEdad(int edad_persona){
    if (EsCorrectaEdad(edad_persona))
        edad = edad_persona;
}

void SetAltura(int altura_persona){
    if (EsCorrectaAltura(altura_persona))
        altura = altura_persona;
}

};

};
```

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.

Retomamos el ejemplo de la página 338. Habíamos definido estas funciones:

```
bool EsMayorEdad (int edad){  
    return edad >= 18;  
}  
  
bool EsAlta (int altura, int edad){  
    if (EsMayorEdad(edad))  
        return altura >= 190;  
    else  
        return altura >= 175;  
}  
  
int main(){  
    ....  
    es_mayor_edad = EsMayorEdad(48);  
    es_alta      = EsAlta(186, 48);
```

¿Cómo quedaría si trabajásemos con una clase Persona?

```
class Persona{  
private:  
    string nombre;  
    int edad;  
    int altura;  
    ....
```

Bastaría añadir los siguientes métodos:

```
class Persona {  
  
private:  
    string nombre;  
    int edad;  
    int altura;  
    .....  
public:  
    .....  
  
    bool EsMayorEdad() {  
        return (edad >= 18);  
    }  
  
    bool EsAlta() {  
        if (EsMayorEdad())  
            es_alta = (altura >= 190);  
        else  
            es_alta = (altura >= 175);  
        return (es_alta);  
    }  
};  
  
int main(){  
  
    Persona una_persona("Manuel", 48, 186);  
    .....  
    bool es_mayor_edad = una_persona.EsMayorEdad();  
    bool es_alta      = una_persona.EsAlta();
```

Observe la diferencia:

▷ Con funciones:

```
.....  
bool es_mayor_edad = EsMayorEdad(48);  
bool es_alta      = EsAlta(186, 48);
```

▷ Con una clase:

```
Persona una_persona("Manuel", 48, 186);  
.....  
bool es_mayor_edad = una_persona.EsMayorEdad();  
bool 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.

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);
```

Sólo incluiremos como datos miembro aquellas características esenciales que determinan una entidad.

La información adicional que se necesite para ejecutar un método, se proporcionará a través de los parámetros de dicho método.

IMPORTANT

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.



V.7.2. Principio de Responsabilidad única y cohesión de una clase

¿Qué preferimos: pocas clases que hagan muchas cosas distintas o más clases que hagan cosas concretas?

Por supuesto, las segundas.

Principio en Programación Dirigida a Objetos.

*Principio de Responsabilidad Única
(Single Responsibility Principle)*

Un objeto debería tener una única responsabilidad, la cual debe estar completamente encapsulada en la definición de su clase.



A cada clase le asignaremos una única responsabilidad. Esto facilitará:

- ▷ La reutilización en otros contextos
 - ▷ La modificación independiente de cada clase (o al menos paquetes de clases)
-

Heurísticas:

- ▷ Si para describir el propósito de la clase se usan más de 20 palabras, puede que tenga más de una responsabilidad.
- ▷ Si para describir el propósito de una clase se usan las conjunciones *y* u *o*, seguramente tiene más de una responsabilidad.

Ejemplo. ¿Es correcto este diseño?:

```
class CuentaBancaria{  
private:  
    double saldo;  
    string identificador;  
public:  
    CuentaBancaria(double saldo_inicial){  
        saldo = saldo_inicial;  
    }  
    string GetIdentificador(){  
        return identificador;  
    }  
    void SetIdentificador(string identificador_cuenta){  
        identificador = identificador_cuenta;  
    }  
    double Saldo(){  
        return saldo;  
    }  
    void ImprimeSaldo(){  
        cout << "Saldo = " << saldo << endl;  
    }  
    void ImprimeIdentificador(){  
        cout << "Identificador = " << identificador << endl;  
    }  
};
```

La responsabilidad de esta clase es gestionar los movimientos de una cuenta bancaria **Y** comunicarse con el dispositivo de salida por defecto. Esto viola el principio de única responsabilidad. Esta clase no podrá reutilizarse en un entorno de ventanas, en el que cout no funciona.



Mezclar E/S y cómputos en una misma clase viola el principio de responsabilidad única. Lamentablemente, ésto se hace con mucha frecuencia, incluso en multitud de libros de texto.

Jamás mezclaremos en una misma clase métodos de **cómputo** con métodos que accedan a los dispositivos de entrada/salida.

IMPORTANT

Una buena solución es implementar métodos tipo *ToString*

```
string ToStringSaldo(){
    return "Saldo = " + to_string(saldo);
}

void ToStringIdentificador(){
    return "Identificador = " + identificador;
}
```

o mejor aún, un método *ToString* que se aplica al objeto *global*:

```
string ToString(){
    string cad;
    cad = "Identificador = " + identificador + "\n";
    cad = cad + Saldo = " + to_string(saldo)+ "\n";
    return cad;
}
```

Tenga en cuenta que no tiene que utilizar todos los campos obligatoriamente sino que participan aquellos que son los representativos o útiles para el propósito de este método.

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_cliente (27,2,1987);  
    nacimiento_cliente.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_cliente (27,2,1987);  
    cout << nacimiento_cliente.ToString();
```



Un concepto ligado al principio de responsabilidad única es el de **cohesión (cohesion)**, que mide cómo están las funciones desempeñadas por un componente software (paquete, módulo, clase o subsistema en general)

Por normal general, cuantos más métodos de una clase utilicen todos los datos miembros de la misma, mayor será la cohesión de la clase.

Ejemplo. ¿Es correcto este diseño?

```
ClienteCuentaBancaria
- double saldo
- string identificador
- string nombre
- string dni
- int dia_ncmto
- int mes_ncmto
- int anio_ncmto
- 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.

V.7.3. Funciones vs Clases

¿Cuándo usaremos funciones y cuándo clases?

- ▷ Como norma general, usaremos clases.
- ▷ Usaremos funciones para implementar tareas muy genéricas que operan sobre tipos de datos simples y que preveamos se pueden utilizar en programas distintos.

```
bool    SonIguales(double uno, double otro);  
bool    EsPrimo(int dato)  
string  ToMayusculas(string cadena);  
double  Redondea(double numero, int decimales);  
.....
```

Ejemplo. Una clase `Fraccion` podría tener el siguiente esquema (los valores los pasamos en el constructor y no se permite su posterior modificación)

Fraccion	
- int	numerador
- int	denominador
+	Fraccion(int el Numerador, int el Denominador)
+ int	Numerador()
+ int	Denominador()
+ void	Normaliza()
+ double	Division()
+ string	ToString()

```
int main(){
    Fraccion una_fraccion(4, 10);      // 4/10

    una_fraccion.Normaliza();          // 2/5
    cout << una_fraccion.ToString();
    ....
```

Para implementar el método `Normaliza`, debemos dividir numerador y denominador por el máximo común divisor. ¿Dónde lo implementamos? El cálculo del MCD no está únicamente ligado a la clase `Fraccion`, por lo que usamos una función global.

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

int MCD(int primero, int segundo){
    .....
}

class Fraccion{
    .....
    void Normaliza(){
        int mcd;

        mcd = MCD(numerador, denominador);
        numerador = numerador/mcd;
        denominador = denominador/mcd;
    }
};

int main(){
    Fraccion una_fraccion(4, 10);      // 4/10

    una_fraccion.Normaliza();          // 2/5
    cout << una_fraccion.ToString();
    .....
```

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.

V.7.4. Comprobación y tratamiento de las precondiciones

En la página 354 se vieron las precondiciones de una función. Las precondiciones se han empleado para establecer la validez de los campos de una clase y asegurar la corrección de los métodos de una clase.

- ▷ ¿Debemos comprobar dentro de la función/método si se satisfacen sus precondiciones?
- ▷ Si decidimos comprobarlo, cómo actuar si se violan?

Ejemplo. Función factorial.

Versión en la que no se comprueba la precondición en la función:

```
// PRE: 0 <= n <= 20
long long Factorial (int n){
    long long fact = 1;
    for (int i = 2; i <= n; i++)
        fact = fact * i;
    return fact;
}
```

Deberemos comprobar la validez del parámetro real antes de la llamada:

```
do {
    cout << "Valor para calcular su factorial: ";
    cin >> valor;
    while (valor < 0 || valor > 20);

    // En este punto 0 <= valor <= 20

    cout << Factorial (n) << endl;
```

Versión en la que se comprueba la precondición:

```
// PRE: 0 <= n <= 20
long long Factorial (int n){
    long long fact = 1;

    if (n >= 0 && n <= 20){
        for (int i = 2; i <= n; i++)
            fact = fact * i;
    }
    return fact;
}
```

Si no se cumple la precondición y se llama a la función se devuelve el valor 1 ¿es lo que queremos? ¿cómo sabemos en la función que hace la llamada que el cálculo es correcto?

En la versión que no comprueba la precondición dentro de la función, si se viola la precondición y se llama a la función el resultado es un desbordamiento aritmético, devolviendo un valor indeterminado.

En la versión que comprueba la precondición dentro de la función, si se viola la precondición y se llama a la función simplemente se evita el cálculo, pero la función devuelve 1.

En este ejemplo, recomendamos omitir la comprobación dentro de la función. En general éste será el procedimiento a seguir.

- ▷ ***Se delega la responsabilidad al usuario de la función.***
- ▷ ***la función queda sencilla y directa.***

Ejemplo. Clase CuentaBancaria.

Versión en la que no se comprueba la precondición:

```
class CuentaBancaria{  
    ....  
    // PRE: cantidad > 0  
    void Ingresa(double cantidad){  
        saldo = saldo + cantidad;  
    }  
};
```

Versión en la que se comprueba la precondición:

```
class CuentaBancaria{  
    ....  
    // PRE: cantidad > 0  
    void Ingresa(double cantidad){  
        if (cantidad > 0)  
            saldo = saldo + cantidad;  
    }  
};
```

En el caso de un ingreso con una cantidad *negativa*, la versión que comprueba la precondición *dentro* de la función *no* modifica el saldo. Está bien, no debería modificarse, pero ¿no sería mejor comprobar *antes de la llamada* que la cantidad a ingresar sea correcta? Definitivamente sí, es nuestra elección.

¿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, independientemente de que se deba hacer antes de la llamada.*

En otro caso, puede omitirse (sobre todo si prima la eficiencia)

Ejemplo. Secuencia de caracteres. Recuerde el ejemplo de la página 515.

```
class SecuenciaCaracteres {  
  
private:  
    static const int TAMANIO = 50;  
    char vector_privado[TAMANIO];  
    int total_utilizados; // PRE: 0 <= total_utilizados <= TAMANIO  
public:  
    SecuenciaCaracteres() : total_utilizados(0) {}  
  
    // PRE: total_utilizados < TAMANIO  
    void Aniade(char nuevo){  
        if (total_utilizados < TAMANIO){  
            vector_privado[total_utilizados] = nuevo;  
            total_utilizados++;  
        }  
    }  
    int TotalUtilizados () {  
        return total_utilizados;  
    }  
    ....  
};  
  
int main(){  
  
    SecuenciaCaracteres secuencia; // <x,x,x,x,x,...,x>, 0  
  
    secuencia.Aniade('t'); // <t,x,x,x,x,...,x>, 1  
    secuencia.Aniade('e'); // <t,e,x,x,x,...,x>, 2  
    secuencia.Aniade('c'); // <t,e,c,x,x,...,x>, 3  
    secuencia.Aniade('a'); // <t,e,c,a,x,...,x>, 4  
    ....
```

Cuando se llena el objeto SecuenciaCaracteres se cumple que total_utilizados == TAMANIO

¿Qué ocurre si se intenta añadir algún carácter más?

Si se llama al método Aniade sin comprobar previamente la precondición, en el método se comprueba si hay algún hueco:

```
if (total_utilizados < TAMANIO)
```

y si no lo hay no se añade ningún elemento, por lo que se evita acceder a una casilla no reservada.

Podríamos comprobar el resultado de la operación -fallida- ya que si después de llamar a Aniade el número de casillas ocupadas (método TotalUtilizados) devuelve el mismo valor que antes de la llamada, significa que no ha sido posible la adición.

Así, en este caso no estaría de más añadir un cortafuegos dentro del método, para garantizar la seguridad de la operación.

Supongamos que decidimos comprobar las precondiciones de una función/método y resulta que no se satisfacen ¿Qué hacemos?

- ▷ **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  
    }  
}
```

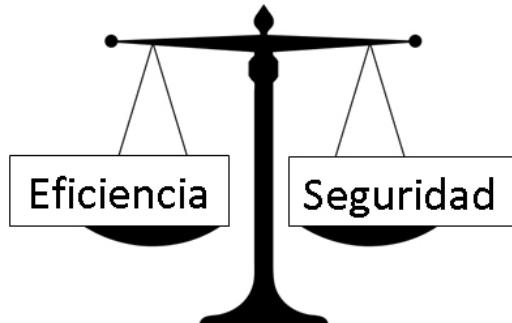
- ▷ **Realizar una acción por defecto.**

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.



¿Comprobar Precondiciones dentro del método/función?

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.

En aras de la claridad y eficiencia de las funciones/métodos es preferible establecer claramente las precondiciones y construir la función pensando que los parámetros serán correctos.

La responsabilidad se delega a la función/método que realice la llamada, que tendrá que asegurarse que se cumplen las precondiciones.

Otras alternativas (que no emplearemos).

¿Y cómo sabemos que se ha violado la precondición ? ¿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
- ▷ Mucho mejor: devolver una excepción

Hasta conocer cómo se gestionan las excepciones veamos la notificación devolviendo un código de error:

- ▷ Si trabajamos con un método/función `void`, lo convertimos en un método/función que devuelva un código de error. El tipo devuelto será `bool` si sólo notificamos la existencia o no del error. Si hay varios posibles errores, lo mejor es usar un enumerado.
- ▷ Si el método/función no es un `void`, podemos usar el mismo valor devuelto para albergar el código de error, pero éste no debe ser uno de los valores legales de los devueltos por el método/función.
- ▷ **MUY IMPORTANTE:** Ninguna de las anteriores soluciones es aplicable a los constructores ya que éstos no pueden devolver valores.

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 dentro del método/función en el que se viola la precondición. Esto también es válido para el constructor.

Si decidimos que es necesario comprobar alguna precondición dentro de un método/función/constructor y ésta se viola, ¿qué debemos hacer como respuesta?

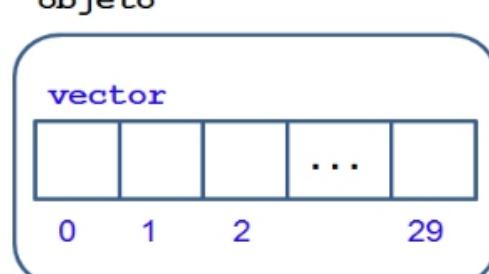
- ▷ *O bien no hacemos nada (salvo lanzar una excepción, tal y como se indica en el siguiente párrafo) o bien realizamos una acción por defecto. La elección apropiada depende de cada problema, aunque lo usual será no hacer nada.*
- ▷ *Además de lo anterior, si decidimos que es necesario notificar dicho error al cliente que ejecuta el método/función podemos devolver un código de error o, mucho mejor, lanzar una excepción.*
En el caso del constructor, sólo podemos lanzar una excepción.
La notificación jamás se hará imprimiendo un mensaje en pantalla desde dentro del método/función.

V.8. Vectores y clases

V.8.1. Los vectores como datos miembro

- ▷ 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 con una constante estática.
En lo que sigue fomentaremos el uso de constantes estáticas ya que i) se definen en la clase y ii) no forman parte del objeto (todos los objetos de la clase acceden y comparten la constante).
- ▷ La idea es que el vector se crea cuando se crea el objeto, como parte de él. Como cualquier dato miembro, el vector existirá mientras exista el objeto.

```
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  
    ....  
}
```



The diagram illustrates an object named 'objeto' represented by a rounded rectangle. Inside it, there is a box labeled 'vector'. Below the 'vector' box is a horizontal array of 30 empty boxes, representing the memory allocation for the vector. The indices 0, 1, 2, ..., 29 are written below the first few elements of the array.

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 sólo permitimos añadir.

Ejercicio. Retomamos el ejemplo del autobús de la página 386 y construimos la clase Autobus

- ▷ El nombre del conductor será un *parámetro* del 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	MAX_PLAZAS
- string	pasajeros[MAX_PLAZAS]
- int	numero_actual_pasajeros
+ Autobus	(string nombre_conductor)
+ void	Situa(int asiento, string nombre_pasajero)
+ string	Pasajero(int asiento)
+ string	Conductor()
+ bool	Ocupado(int asiento)
+ int	Capacidad()
+ int	NumeroActualPasajeros()

Esta clase se usará en la siguiente forma:

```
int main(){

    string conductor;
    .....
    // Leer conductor
    .....
    Autobus ruta044-003 (conductor);
    .....
    while (hay_datos_por_leer){
        ruta044-003.Situa(asiento, nombre_pasajero);
        .....
    }
}
```

Veamos el programa completo:

```
int main(){

    const string TERMINADOR = "*";
    string conductor, nombre_pasajero;
    int     asiento, capacidad_autobus;

    // Para crear un "Autobus" se requiere el nombre del conductor
    cout << "Introduzca nombre del conductor: ";
    getline (cin, conductor);
    cout << endl;

    Autobus ruta044-003(conductor); // Constructor
    // ruta044-003 está vacío (numero_actual_pasajeros es 0)

    cout << "Introduzca los nombres de los pasajeros y su asiento."
        << "Termine con " << TERMINADOR << endl << endl;

    // Lectura adelantada del nombre del pasajero
    cout << "Nombre pasajero: ";
    getline (cin, nombre_pasajero);

    while (nombre_pasajero != TERMINADOR){

        // Lectura segura del número de asiento
        do {
            cout << "Asiento: ";
            cin >> asiento;
        } while ((asiento<1) || (asiento>MAX_PLAZAS) || (Ocupado(asiento)));

        // El número de asiento es correcto: actualizar el "Autobus"
        ruta044-003.Situa(asiento, nombre_pasajero);
    }
}
```

```
// Nueva lectura del nombre del pasajero
cout << "Nombre pasajero: " << endl;
getline (cin, nombre_pasajero);

}

// Mostrar el estado del objeto "Autobus"
cout << endl;
cout << "Conductor: " << ruta044-003.Conductor() << endl;
cout << "Total de pasajeros: "
    << ruta044-003.NumeroActualPasajeros() << endl;
cout << "Pasaje: " << endl;

capacidad_autobus = ruta044-003.Capacidad(); // Consulta

for (int i=1; i < capacidad_autobus; i++){

    cout << "Asiento número: " << i;
    // Consultar estado y en su caso, obtener un valor
    if (ruta044-003.Ocupado(i))
        cout << " Pasajero: " << ruta044-003.Pasajero(i);
    else cout << " Vacío";
    cout << endl;
}
}
```

Implementamos ahora la clase. Observe cómo ocultamos información al situar dentro de la clase las constantes MAX_PLAZAS y VACIO.

```
class Autobus {

private:
    const string VACIO = "";    // Asiento vacío
    static const int MAX_PLAZAS = 50; // Capacidad (viajeros)
    string pasajeros[MAX_PLAZAS+1] = {VACIO};
    int numero_actual_pasajeros; // Casillas ocupadas
```

```
public:

    Autobus (string nombre_conductor) {

        pasajeros[0] = nombre_conductor;
        numero_actual_pasajeros = 0; // El conductor no es pasajero

        for (int i=1; i < MAX_PLAZAS; i++)
            pasajeros[i] = VACIO;
    }

    int Capacidad () {
        return MAX_PLAZAS;
    }

    int NumeroActualPasajeros (){
        return numero_actual_pasajeros;
    }

    string Conductor () {
        return pasajeros[0];
    }

    // PRE: 1 <= asiento <= MAX_PLAZAS
    string Pasajero (int asiento){
        return pasajeros[asiento];
    }

    // PRE: 1 <= asiento <= MAX_PLAZAS
    bool Ocupado (int asiento){
        return pasajeros[asiento] != VACIO;
    }
}
```

```
// PRE: 1 <= asiento <= MAX_PLAZAS
// PRE: !Ocupado (asiento)

void Situa (int asiento, string nombre_pasajero) {

    if (asiento >= 1 && asiento <= MAX_PLAZAS) {

        if (! Ocupado(asiento)) {
            pasajeros[asiento] = nombre_pasajero;
            numero_actual_pasajeros++;
        }
    }
}

};

};
```

Ampliación:



El método `Situa` podría devolver un valor `bool` indicando si se ha podido situar correctamente al pasajero. Si queremos notificar problemas durante la ejecución de un método es mejor hacerlo con el mecanismo de las excepciones.

En cualquier caso, recuerde que sugerimos y fomentamos el uso de precondiciones. Esta estrategia puede reforzarse en aras de la seguridad con comprobaciones adicionales dentro de la función.

Este es el caso del método `Situa` en el que se escribe el cortafuegos que marcan las precondiciones para evitar acceder a una casilla inadecuada (por posición fuera de índices o estado incorrecto).

Ejemplo. Retomamos el ejemplo de la página 384 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 (los huecos quedan al final).

utilizados = 4								
notas	X X X X				$?$	$?$	\dots	$?$
nombres	X X X X				$?$	$?$	\dots	$?$

Para forzar esta situación, la única forma de insertar nuevos datos será a través del siguiente método:

```
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.

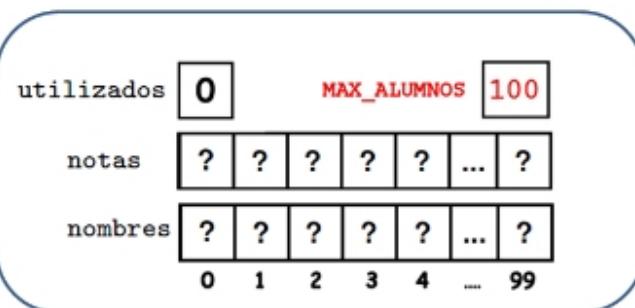
CalificacionesFinales	
-	const int MAX_ALUMNOS
-	double notas[MAX_ALUMNOS]
-	string nombres[MAX_ALUMNOS]
-	int utilizados
+	CalificacionesFinales()
+	Capacidad()
+	TotalAlumnos()
+	CalificacionAlumno(int posicion)
+	NombreAlumno(int posicion)
+	Aniade(string nombre_alumno, double nota_alumno)
+	MediaGlobal()
+	SuperanMedia()

Esta clase se usará en la siguiente forma:

```
notas_FP

int main(){
    CalificacionesFinales notas_FP;
    .....

    while (hay_datos_por_leer){
        .....
        notas_FP.Aniade(nombre, nota);
    }
}
```



Veamos el programa completo:

```
class CalificacionesFinales {

private:
    static const int MAX_ALUMNOS = 100; // Capacidad
    double notas[MAX_ALUMNOS]; // Las calificaciones
    string nombres[MAX_ALUMNOS]; // Los nombres
    int utilizados; // Número de casillas usadas
```

```
public:

    // Constructor: objeto vacío
    CalificacionesFinales() : utilizados(0) { }

    // Métodos de consulta de la capacidad y de la ocupación real

    int Capacidad() {
        return MAX_ALUMNOS;
    }

    int TotalAlumnos() {
        return utilizados;
    }

    // Métodos para consulta de contenido

    double CalificacionAlumno (int posicion){
        return notas[posicion];
    }

    string NombreAlumno (int posicion){
        return nombres[posicion];
    }

    // Método para actualizar (añadir) contenido

    // PRE: 0 <= utilizados < MAX_ALUMNOS
    void Aniade (string nombre_alumno, double nota_alumno){
        if (0<= utilizados && utilizados < MAX_ALUMNOS){
            notas[utilizados] = nota_alumno;
            nombres[utilizados] = nombre_alumno;
            utilizados++;
        }
    }
}
```

```
// Métodos de cálculo

// PRE: utilizados > 0
double MediaGlobal () {
    double suma = 0;
    for (int i = 0; i < utilizados; i++)
        suma += notas[i];
    return suma/utilizados;
}

// PRE: utilizados > 0
int SuperanMedia (){
    double media = MediaGlobal(); // Precondición común
    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 = "*";
    string nombre;
    double nota;

    CalificacionesFinales notas_FP; // Constructor: objeto vacío

    // Lectura de la capacidad (máximo número de casillas) para evitar
    // tener que consultar repetidamente este valor (no cambia)
    int capacidad = notas_FP.Capacidad();
```

```
// Lectura de datos

cout << "Nombre de alumno (" << TERMINADOR << " para terminar): ";
getline (cin, nombre);

while (nombre != TERMINADOR && notas_FP.TotalAlumnos() < capacidad){

    cout << "Nota: ";
    cin >> nota;

    notas_FP.Aniaude(nombre, nota);

    cout << "Nombre de alumno (" << TERMINADOR << " para terminar): ";
    getline (cin, nombre);
}

// Calculos

double media = notas_FP.MediaGlobal();
int superan_media = notas_FP.SuperanMedia();

// Resultados

cout << "Cálculos sobre " << notas_FP.TotalAlumnos()
    << " alumnos." << endl;

cout << "Media Aritmetica = " << media << endl;
cout << superan_media << " alumnos han superado la media" << endl;

return 0;
}
```

V.8.2. Comprobación de las precondiciones

Dentro de un método, ¿debemos comprobar siempre que se satisfacen sus precondiciones? Depende. La comprobación supone una *carga computacional adicional*. Sin embargo, puede evitar errores graves

Decimos *carga adicional* porque asumimos que ya se han comprobado *antes* de la llamada. Anteriormente hemos hecho referencia a esta comprobación adicional como *cortafuegos*.

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.

Este consejo se aplica como caso especial en el caso de métodos que modifican componentes de un vector dato miembro de una clase.

Ejemplo. En el ejemplo del autobús, hemos comprobado la precondición 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 precondición:

```
// PRE: 1 <= asiento <= MAX_PLAZAS
// PRE: !Ocupado (asiento)

void Situa (int asiento, string nombre_pasajero){
    if (! Ocupado(asiento)){
        pasajeros[asiento] = nombre_pasajero;
        numero_actual_pasajeros++;
    }
}
```



▷ Versión comprobando la precondición:

```
// PRE: 1 <= asiento <= MAX_PLAZAS
// PRE: !Ocupado (asiento)
void Situa (int asiento, string nombre_pasajero){

    if (asiento >= 1 && asiento <= MAX_PLAZAS){
        if (! Ocupado(asiento)){
            pasajeros [asiento] = nombre_pasajero;
            numero_actual_pasajeros++;
        }
    }
}
```



En cambio, en el método Pasajero si el índice que se proporciona no es correcto, simplemente devolverá un valor basura, pero no corremos el peligro de modificar una componente no reservada. Por lo tanto, podríamos evitar la comprobación de la precondición *dentro* del método. Lo mismo ocurre con el método Ocupado.

En cualquier caso, *siempre comprobaremos/nos aseguraremos de que se cumplen las precondiciones antes de la llamada a un método*.

Siempre comprobaremos/nos aseguraremos de que se cumplen las precondiciones antes de la llamada a un método.

Ejemplo. En el ejemplo de las notas, hemos comprobado la precondición 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 precondición:

```
// PRE: utilizados < MAX_ALUMNOS  
  
void Aniade(string nombre_alumno, double nota_alumno) {  
    notas[utilizados] = nota_alumno;  
    nombres[utilizados] = nombre_alumno;  
    utilizados++;  
}
```



▷ Versión comprobando la precondición:

```
// PRE: utilizados < MAX_ALUMNOS  
  
void Aniade (string nombre_alumno, double nota_alumno) {  
    if (utilizados < MAX_ALUMNOS) {  
        notas[utilizados] = nota_alumno;  
        nombres[utilizados] = nombre_alumno;  
        utilizados++;  
    }  
}
```



Al no modificar componentes, hemos decidido no comprobar las precondiciones en los métodos de consulta `CalificacionAlumno` y `NombreAlumnno`

V.8.3. Vectores como datos locales de un método

- ▷ Un vector puede ser un *dato local* de un método.
- ▷ Debería declararse con una constante (para la capacidad) definida dentro del método o 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 376

```
class MiClase{  
    .....  
    void Metodo(){  
        const int TAMANIO = 30; // static cuando es dato miembro  
        char vector[TAMANIO]; // (no es el caso)  
        .....  
    }  
};  
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 tareas.

No puede devolverse un vector, aunque sí podremos devolver objetos que contienen vectores.

Ejemplo. Clase Fecha (simple):

```
class Fecha
{
private:
    ...
    int dia, mes, anio;

public:
    .....

    // Devuelve: 1 (si es Lunes), 2 (si es Martes) ...
    int GetNumDiaDeLaSemana() {
        .....
    }

    string GetNombreDiaDeLaSemana () {
        string dia_semana_esp []= {"-", "Lunes", "Martes", "Miercoles",
                                   "Jueves", "Viernes", "Sabado", "Domingo"};
        return (dia_semana_esp[GetNumDiaDeLaSemana()]);
    }
    .....
};

int main() {

    Fecha fecha_hoy; // Constructor sin parámetros

    cout << "Hoy es: " << fecha_hoy.GetNombreDiaDeLaSemana() << ", "
        << fecha_hoy.ToString() << endl;
    .....
}
```

V.8.4. 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 modificar el contenido del vector.

Es posible que esa fuera nuestra intención, pero rompe el concepto de paso de parámetro por valor.

- ▷ El paso de vectores a funciones debería hacerse cuando se conoce la gestión de memoria en C++ usando punteros. De esta manera dominará las técnicas necesarias y sabrá qué está haciendo en una función.

El uso de punteros va más allá del ámbito de esta asignatura.

- ▷ 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.

Potenciaremos la construcción de clases genéricas (colecciones o secuencias) que encapsulen los datos (el contenido) y las propiedades de la secuencia (capacidad y número de casillas usadas). Los objetos de estas clases podrán pasarse a funciones y devueltas por éstas.

En esta asignatura no pasaremos los vectores como parámetros a funciones o métodos.

V.8.5. Vectores de objetos

Para crear un vector de objetos, se usa la notación habitual:

```
const int TAMANIO = 50;  
MiClase vector_de_objetos[TAMANIO];
```

Recordemos que el compilador crea un objeto cuando se declara:

```
class MiClase {  
    .....  
};  
int main() {  
    MiClase objeto;  
    .....  
}
```

ya que proporciona de oficio un constructor sin argumentos.

Lo mismo ocurre con las componentes de un vector; se crean en el momento de definir el vector:

```
class MiClase{  
    .....  
};  
int main() {  
    const int TAMANIO = 50;  
    MiClase vector_de_objetos[TAMANIO]; // <- Se crean 50 objetos  
    .....  
}
```

Cada casilla de `vector_de_objetos` contiene un objeto de la clase `MiClase`. Cada uno de esos objetos debe crearse, y para ello se requiere un **constructor** (sin parámetros).

Por lo tanto, si se va a definir un vector de objetos de una clase, ésta debe proporcionar un **constructor sin parámetros**.

- ▷ O bien porque no tenga ningún otro constructor definido, y sea el proporcionado de oficio por el compilador.
- ▷ O bien porque esté definido explícitamente en la clase.

Si la clase no tiene constructor sin parámetros, hay que crear los objetos en la misma declaración del vector. No es una solución muy aconsejable (por poco práctica y/o generalizable). Ejemplo:

```
class MiClase {  
public:  
    MiClase (int parametro){  
        .....  
    }  
};  
int main(){  
    const int TAMANIO = 2;  
  
    MiClase vector_de_objetos[TAMANIO]; // Error de compilación   
  
    MiClase un_objeto(5);  
    MiClase otro_objeto(6);  
    MiClase vector_de_objetos[TAMANIO] = {un_objeto, otro_objeto};  
    .....  
}
```

Cuando se define un vector de objetos, se crean automáticamente tantos objetos como capacidad tenga el vector. Si trabajamos con un vector de objetos, la clase debe proporcionar un constructor sin parámetros.

V.9. La clase Secuencia de caracteres

V.9.1. Métodos básicos

Vamos a construir una clase genérica `SecuenciaCaracteres` que nos servirá de base para ilustrar distintas operaciones sobre vectores.

Pretende representar una colección de caracteres que se disponen en casillas consecutivas en un vector, sin dejar huecos, desde el principio. Llamaremos *secuencias* a este tipo de colecciones.

No permitiremos operaciones (como añadir o eliminar componentes) que impliquen dejar huecos.

Internamente, usaremos un vector (recuerde el ejemplo de las notas), es decir:

```
total_utilizados = 4
```

X	X	X	X	?	?	...	?
---	---	---	---	---	---	-----	---

Una primera versión:

SecuenciaCaracteres	
- const int	TAMANIO
- 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)

- ▷ Capacidad **devuelve el tamaño del vector (número de componentes reservadas en memoria)**
- ▷ TotalUtilizados **devuelve el número de componentes ocupadas.**
- ▷ Aniade **añade (al final) un valor adicional. Modifica -incrementa- el número de componentes ocupadas.**
- ▷ Modifica **cambia el valor de una componente ya existente (previamente añadida a través del método Aniade).**
- ▷ Elemento **devuelve el valor de la componente en el índice señalado.**

Los métodos Aniade y Modifica **pueden modificar el contenido del vector.** El método Elemento **accede a una casilla determinada.** Para impedir que estos métodos accedan a casillas no reservadas, estableceremos *pre-condiciones* que impidan i) añadir un elemento si no queda espacio y ii) acceder a una posición incorrecta para modificar o consultar su contenido.

Un ejemplo de uso de esta clase:

```
int main(){

    SecuenciaCaracteres secuencia; // <>, 0 --> secuencia vacía

    secuencia.Anade('h');      // -> <h>, 1
    secuencia.Anade('o');      // -> <h,o>, 2
    secuencia.Anade('l');      // -> <h,o,l>, 3
    secuencia.Anade('a');      // -> <h,o,l,a>, 4

    // Número de casillas ocupadas (con datos válidos)
    int num_elementos = secuencia.TotalUtilizados(); // 4

    for (int i = 0; i < num_elementos; i++)
        cout << secuencia.Elemento(i) << " ";

    // h o l a

    secuencia.vector_privado[1] = 'g'; // Error compilac. Dato privado. ☺
    secuencia.vector_privado[70] = 'g'; // Error compilac. Dato privado.

    secuencia.Modifica(9,'l');      // -> <h,o,l,a>, 4
    // Acceso incorrecto --> No se ha hecho nada

    secuencia.Modifica(0,'L');      // -> <L,o,l,a>, 4
    .....
```

Definamos los métodos:

```
class SecuenciaCaracteres {  
  
private:  
    static const int TAMANIO = 50;    // Número de casillas a reservar  
    char vector_privado[TAMANIO];    // Vector de datos  
    int total_utilizados;           // Número de casillas usadas  
    // PRE: 0 <= total_utilizados < = TAMANIO  
  
public:  
    // Constructor sin argumentos. Crea una secuencia vacía.  
    SecuenciaCaracteres() : total_utilizados(0) { }  
  
    // Métodos de consulta  
    // Devuelve el número de componentes ocupadas (datos válidos)  
    int TotalUtilizados() {  
        return total_utilizados;  
    }  
    // Devuelve el tamaño del vector (num. casillas reservadas)  
    int Capacidad() {  
        return TAMANIO;  
    }  
  
    // Añade (al final) un valor adicional.  
    // Modifica -incrementa- el número de componentes ocupadas.  
    // PRE: total_utilizados < TAMANIO  
    void Aniade (char nuevo) {  
        if (total_utilizados < TAMANIO){  
            vector_privado[total_utilizados] = nuevo;  
            total_utilizados++;  
        }  
    }  
}
```

```
// Cambia el valor de una componente ya existente.  
// PRE: 0 <= posicion < total_utilizados  
void Modifica (int posicion, char nuevo) {  
    if (posicion >= 0 && posicion < total_utilizados)  
        vector_privado[posicion] = nuevo;  
}  
  
// Devuelve el valor de una componente del vector.  
// PRE: 0 <= indice < total_utilizados  
char Elemento (int indice) {  
    return vector_privado[indice];  
}  
};
```

¿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{  
    .....  
    // PRE: 0 <= nuevo_total <= total_utilizados  
    void Trunca (int nuevo_total) {  
        if (0 <= nuevo_total && nuevo_total <= total_utilizados)  
            total_utilizados = nuevo_total;  
    }  
};
```



Observe que si nuevo_total es 0, el resultado es el mismo que EliminaTodos y si nuevo_total es total_utilizados no hay ningún cambio.

Si queremos una secuencia de reales:

```
class SecuenciaReales {  
  
private:  
    static const int TAMANIO = 50; // Número de casillas a reservar  
    double vector_privado[TAMANIO]; // Vector de datos  
    int total_utilizados; // Número de casillas usadas  
    // PRE: 0 <= total_utilizados < = TAMANIO  
  
public:  
    SecuenciaReales () : total_utilizados(0) {}  
  
    // Los métodos TotalUtilizados() y Capacidad() no varían  
  
    // PRE: total_utilizados < TAMANIO  
    void Aniade (double nuevo) {  
        if (total_utilizados < TAMANIO){  
            vector_privado[total_utilizados] = nuevo;  
            total_utilizados++;  
        }  
    }  
    // PRE: 0 <= posicion < total_utilizados  
    void Modifica (int posicion, double nuevo) {  
        if (posicion >= 0 && posicion < total_utilizados)  
            vector_privado[posicion] = nuevo;  
    }  
    // PRE: 0 <= indice < 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. 

V.9.2. Métodos de búsqueda

Veamos cómo quedarían implementados los métodos de búsqueda vistos en la sección IV.3.1 dentro de la clase SecuenciaCaracteres.

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 << "No encontrado";  
    else  
        cout << "Encontrado en la posición " << pos_encontrado;  
}
```

Implementación:

```
class SecuenciaCaracteres {  
  
private:  
    static const int TAMANIO = 50;  
    char vector_privado[TAMANIO];  
    int total_utilizados;  
  
public:  
    .....  
  
    // Devuelve la posición de la primera aparición de "buscado"  
    // en toda la secuencia. Si no existe, devuelve -1  
  
    int PrimeraOcurrencia (char buscado) {  
  
        bool encontrado = false;  
        int i = 0;  
  
        while (i < total_utilizados && !encontrado){  
            if (vector_privado[i] == buscado)  
                encontrado = true; // Terminar la búsqueda  
            else  
                i++; // Seguir buscando  
        }  
  
        return (encontrado ? i : -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` conlleva una recarga computacional adicional. 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, preferimos trabajar dentro de los métodos de la clase accediendo directamente al vector privado sin llamar a ningún método más.

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:  
    .....  
  
    // Devuelve la posición de la primera aparición de "buscado"  
    // entre las posiciones "pos_izda" y "pos_dcha".  
    // Si no existe, devuelve -1.  
    // PRE: 0 <= pos_izda <= pos_dcha < total_utilizados  
  
    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; // Terminar la búsqueda  
            else  
                i++; // Seguir buscando  
  
        return (encontrado ? i : -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);  
}
```

Con los métodos de búsqueda introducidos, la clase SecuenciaCaracteres **nos quedaría como sigue:**

SecuenciaCaracteres	
- const int	TAMANIO
- 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)

Búsqueda binaria

Precondiciones de uso: Se aplica sobre un vector *ordenado*.

Importante: 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. Observe que la llamada (salvo el nombre) es la que conocemos.

Cabecera:

```
class SecuenciaCaracteres{  
    ....  
    int BusquedaBinaria (char buscado) { ..... }  
};
```

Llamada:

```
int main(){  
    SecuenciaCaracteres secuencia;  
    ....  
    pos_encontrado = secuencia.BusquedaBinaria('B');  
  
    if (pos_encontrado == -1)  
        cout << "No encontrado";  
    else  
        cout << "Encontrado en la posición " << pos_encontrado;  
  
}
```

Implementación:

```
class SecuenciaCaracteres {  
private:  
    static const int TAMANIO = 50;  
    char vector_privado[TAMANIO];  
    int total_utilizados;  
public:  
    .....  
    // Devuelve la posición de la primera aparición de "buscado"  
    // en toda las secuencia. Si no existe, devuelve -1.  
    // PRE: La secuencia está ordenada  
  
    int BusquedaBinaria (char buscado) {  
        int izda = 0;  
        int dcha = total_utilizados - 1;  
        int centro = (izda + dcha) / 2;  
        bool encontrado = false;  
  
        while (izda <= dcha && !encontrado){  
            if (vector_privado[centro] == buscado) encontrado = true;  
            else {  
                if (buscado < vector_privado[centro])  
                    dcha = centro - 1;  
                else  
                    izda = centro + 1;  
            }  
            centro = (izda + dcha) / 2;  
        }  
        return (encontrado ? i : -1);  
    }  
    .....  
};
```

Buscar el mínimo

¿Devolvemos el mínimo de la secuencia? 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;  
                }  
            } // for  
  
        } // if (total_utilizados > 0)  
  
        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{  
    ....  
    // PRE: 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:  
    .....  
    // PRE: 0 <= izda <= dcha < total_utilizados  
    int PosicionMinimoEntre (int izda, int dcha) {  
  
        char minimo = vector_privado[izda];  
        int 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);  
}
```

Con las nuevas adiciones tenemos:

SecuenciaCaracteres	
- const int	TAMANIO
- 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)

Inserción de un valor

La inserción de un valor dentro del vector implica desplazar las componentes que hay a su derecha. Debemos comprobar que los accesos son correctos para evitar modificar componentes no reservadas.

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:  
    .....  
  
    // PRE: total_utilizados < TAMANIO  
    // PRE: 0 <= pos_insercion <= total_utilizados  
  
    void Inserta (int pos_insercion, char valor_nuevo) {  
  
        if (total_utilizados < TAMANIO && pos_insercion >= 0 &&  
            pos_insercion <= total_utilizados) {  
  
            // Desplazar hacia la derecha  
            for (int i = total_utilizados ; i > pos_insercion ; i--)  
                vector_privado[i] = vector_privado[i-1];  
  
            // Escribir en la posicion "pos_insercion"  
            vector_privado[pos_insercion] = valor_nuevo;  
            total_utilizados++;  
        }  
    }  
    .....  
}
```

Eliminación de un valor

Debemos pasar como parámetro la posición a eliminar.

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:  
    .....  
    // PRE: 0 <= posicion < total_utilizados  
  
    void Elimina (int posicion) {  
  
        if (posicion >= 0 && posicion < total_utilizados) {  
  
            // Desplazar hacia la izquierda  
  
            int tope = total_utilizados-1;  
  
            for (int i = posicion ; i < tope ; i++)  
                vector_privado[i] = vector_privado[i+1];  
  
            // Actualizar tamaño  
            total_utilizados--;  
        }  
    }  
    .....  
};
```

Seguimos añadiendo funcionalidad a la clase:

SecuenciaCaracteres	
- const int	TAMANIO
- 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)

V.9.3. Llamadas entre métodos

Ejercicio. Borre el elemento mínimo de un vector.

¿Construimos el siguiente método?

```
void EliminaMinimo ()
```

Lo normal será que no lo hagamos ya que es mejor definir el método

```
void Elimina (int posicion_a_eliminar)
```

y realizar la siguiente llamada:

```
int main(){
    SecuenciaCaracteres cadena;
    .....
    cadena.Elimina (cadena.PosicionMinimo());
```

A la hora de diseñar la interfaz de una clase (el conjunto de métodos públicos), intente construir un conjunto **mínimo** de operaciones *primitivas* que puedan reutilizarse lo máximo posible.



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.

```
void EliminaMinimo(){
    Elimina (PosicionMinimo());
}
```

V.9.4. Algoritmos de ordenación

Cabecera:

```
class SecuenciaCaracteres{  
    ....  
    void Ordena () { ..... }  
};
```

Llamada:

```
int main(){  
  
    SecuenciaCaracteres secuencia;  
    ....  
    secuencia.Ordena ();
```

Para diferenciar los distintos algoritmos de ordenación usaremos nombres diferentes para cada método.

Ordenación por selección

Implementación:

```
class SecuenciaCaracteres {  
  
private:  
    static const int TAMANIO = 50;  
    char vector_privado[TAMANIO];  
    int total_utilizados;  
  
    // PRE: 0<= pos_izda, pos_dcha < total_utilizados  
    void IntercambiaComponentes (int pos_izda, int pos_dcha) {  
        char intercambia = vector_privado[pos_izda];  
        vector_privado[pos_izda] = vector_privado[pos_dcha];  
        vector_privado[pos_dcha] = intercambia;  
    }  
  
public:  
    ....  
    void Ordena_por_Seleccion () {  
  
        // Buscar el menor entre las casillas "izda" y "total_utilizados"-1  
        // e intercambiar ese valor por el de la posición "izda"  
  
        for (int izda = 0 ; izda < total_utilizados ; izda++){  
            int pos_min = PosicionMinimoEntre (izda, total_utilizados-1);  
            IntercambiaComponentes (izda, pos_min);  
        }  
    }  
    ....  
};
```

Observe que no hemos comprobado las precondiciones en el método privado `IntercambiaComponentes`. Esto lo hemos hecho para aumentar la eficiencia de los métodos públicos que lo usen (por ejemplo, los métodos de ordenación). Si quisieramos ofrecer un método público que hiciese lo mismo, sería conveniente (por seguridad) 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 560 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 () {  
  
        for (int izda = 1; izda < total_utilizados; izda++) {  
  
            char a_insertar = vector_privado[izda];  
  
            // Buscar el sitio que le corresponde a "a_insertar"  
            // desplazando hacia la derecha los valores mayores  
  
            int i = izda;  
            while (i > 0 && a_insertar < vector_privado[i-1]) {  
                vector_privado[i] = vector_privado[i-1];  
                i--;  
            }  
            vector_privado[i] = a_insertar; // Colocar en su lugar  
        } // for  
    }  
    .....  
};
```

Ordenación por burbuja

► Primera Aproximación

```
class SecuenciaCaracteres {  
  
private:  
    static const int TAMANIO = 50;  
    char vector_privado[TAMANIO];  
    int total_utilizados;  
  
    // PRE: 0<= pos_izda, pos_dcha < total_utilizados  
    void IntercambiaComponentes (int pos_izda, int pos_dcha) {  
        char intercambia = vector_privado[pos_izda];  
        vector_privado[pos_izda] = vector_privado[pos_dcha];  
        vector_privado[pos_dcha] = intercambia;  
    }  
  
public:  
    ....  
    void Ordena_por_Burbuja (){  
  
        for (int izda = 0; izda < total_utilizados; izda++)  
            for (int i = total_utilizados-1 ; i > izda ; i--)  
                if (vector_privado[i] < vector_privado[i-1])  
                    IntercambiaComponentes (i, i-1);  
    }  
    ....  
};
```

► Segunda Aproximación

```
class SecuenciaCaracteres {  
  
private:  
    static const int TAMANIO = 50;  
    char vector_privado[TAMANIO];  
    int total_utilizados;  
  
    // PRE: 0<= pos_izda, pos_dcha < total_utilizados  
    void IntercambiaComponentes (int pos_izda, int pos_dcha) {  
        .....  
    }  
  
public:  
    .....  
    void Ordena_por_BurbujaMejorado () {  
  
        bool cambio = true;  
  
        for (int izda = 0; izda < total_utilizados && cambio; izda++) {  
  
            cambio = false;  
  
            for (int i = total_utilizados-1; i > izda; i--)  
                if (vector_privado[i] < vector_privado[i-1]) {  
                    IntercambiaComponentes (i, i-1);  
                    cambio = true;  
                }  
        } // for izda  
    }  
    .....  
};
```

Con la adición de los métodos de ordenación, la clase Secuenciacaracteres quedaría:

SecuenciaCaracteres	
- const int	TAMANIO
- char	vector_privado[TAMANIO]
- int	total_utilizados
- void	IntercambiaComponentes (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()

V.10. La clase string

V.10.1. Métodos básicos

C++ proporciona clases específicas para trabajar de forma segura con vectores de datos como por ejemplo la plantilla `vector` de la STL.

Por otra parte, seguro que se ha dado cuenta que el tipo `string` -que hemos usado con mucha frecuencia- es, realmente es una *clase*, por lo que los datos de tipo `string` son *objetos*. La clase `SecuenciaCaracteres` se ha definido con métodos que recuerdan a los existentes en un `string`:

string	SecuenciaCaracteres
push_back	Aniade
capacity	Capacidad
size	TotalUtilizados
clear	EliminaTodos
at	Elemento

Ampliación:

Otros métodos interesantes sobre datos `string` son:

- `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.
-

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.
- ▷ Puede ser argumento del operador `<<`
- ▷ Acceder a las componentes individuales como si fuese un vector clásico, con la notación de corchetes.
- ▷ 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 `""`

En cualquier caso, les recomendamos que consulten los manuales cuando necesite alguna funcionalidad de un objeto `string`.

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

int main(){
    string cadena;

    if (cadena == "")  
        cout << "Vacía";                                // Vacía  
    cout << endl;

    cadena.push_back('F');  
    cout << cadena << endl;                            // F

    cadena = "Funda";                                    // Funda  
    cadena = cadena + "ment";                          // Fundament  
    cadena = cadena + 'o';                            // Fundamento  
    cout << cadena << endl;                            // Fundamento

    cadena.push_back('s');  
    cout << cadena << endl;                            // Fundamentos  
    cout << cadena.capacity() << endl;                // 20  
    cout << cadena.size() << endl;                      // 11

    cadena[9] = 'a';  
    cout << cadena << endl;                            // Fundamentas  
    cadena[12] = 'z';  
    cout << cadena << endl;                            // Comportamiento indeterminado  
    ....
```



V.10.2. Lectura de un string con getline

¿Cómo leemos una variable de tipo string? cin se salta los separadores:

```
string cadena1, cadena2;

cout << "\nIntroduzca nombre: "; // "Uno      Dos"
cin >> cadena1;
cin >> cadena2;
cout << "-" << cadena1 << "-" << cadena2 << "-"; // -Uno-Dos-
```

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 saltos de línea que pudiera haber:
    do{
        getline(cin, cadena);
    } while (cadena == "");
    return cadena;
}

int main(){
    string cadena1, cadena2;

    cout << "\nIntroduzca nombre: ";
    cadena1 = LeeCadenaHastaEnter(); // Uno      Dos Tres
    cout << "\nIntroduzca nombre: "; // Cuatro    Cinco
    cadena2 = LeeCadenaHastaEnter();
    cout << "-" << cadena1 << "-" << cadena2 << "-";
                                // -Uno      Dos Tres-Cuatro    Cinco-
```

V.10.3. El método ToString

En numerosas ocasiones, será bastante útil proporcionar un método `ToString` a las clases que construyamos. Por ejemplo, para una clase `Autobus`, podría devolver una cadena con los nombres de todos los pasajeros. Para una clase `CalificacionesFinales` podría devolver una cadena con los nombres y notas de todos los alumnos.

En nuestra clase `SecuenciaCaracteres`, está clara su utilidad:

```
class SecuenciaCaracteres {  
    .....  
    string ToString() {  
        string cadena;  
  
        for (int i=0; i < total_utilizados; i++)  
            cadena = cadena + vector_privado[i];  
  
        return cadena;  
    }  
    .....  
};  
  
int main(){  
    SecuenciaCaracteres secuencia;  
    .....  
    cout << secuencia.ToString() << endl;  
}
```

Tema VI

Clases (Segunda parte)

Objetivos:

- ▷ Empezar a diseñar clases que colaboran entre sí para poder resolver problemas más complejos.
- ▷ Saber que los métodos de una clase pueden: a) tener datos locales, b) tener parámetros formales, y c) devolver objetos de otra clase.
- ▷ Saber que una clase puede tener datos miembro que son a su vez objetos de otra clase.
- ▷ Trabajar con la clase Tabla

Autor: Juan Carlos Cubero.

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

VI.1. Métodos y objetos

VI.1.1. Objetos y funciones

En lo que resta de asignatura, no definiremos funciones globales que actúen sobre objetos. Este diseño puede aceptarse sólo en algunos casos excepcionales como ya vimos en la sección V.7.3.

Así pues, nunca definiremos funciones del tipo:

```
class Punto2D{  
    .....  
};  
class Recta{  
    .....  
};  
  
bool Contiene (Recta una_recta, Punto2D un_punto){  
    .....  
}
```

Lo que haremos será *definir un método* de la clase Recta:

```
class Punto2D{  
    .....  
};  
class Recta{  
    .....  
    bool Contiene (Punto2D un_punto){  
        .....  
    }  
};
```



Evite por ahora la definición de funciones globales a las que se les pase objetos como parámetros. Debemos definir métodos que se ejecuten sobre los objetos. Permitiremos el uso de funciones globales cuando éstas sean muy genéricas y sólo se les pase como parámetros tipos simples.

IMPORTANT

VI.1.2. Objetos locales de un método

Dentro de un método podemos declarar objetos

Nos planteamos si en un método puede declararse un objeto (de la misma clase o de otra). La respuesta es **sí**.

- ▷ El objeto se creará en la pila, dentro del marco del método que se está ejecutando, como cualquier otra *variable local*.
- ▷ Para crear el objeto local actuará el *constructor* que se ajuste a la declaración.
- ▷ El objeto existirá mientras se esté ejecutando el método. Cuando termine la ejecución del método, automáticamente se *libera* la memoria asociada al objeto.

```
class UnaClase {  
    ....  
    void UnMetodoUnaClase () {  
        UnaClase un_objeto; // Objeto local (const. sin parámetros)  
        ....  
    }  
};  
class OtraClase{  
    ....  
    int UnMetodoOtraClase () {  
        UnaClase obj1; // Objeto local (const. sin parámetros)  
        UnaClase obj2 (int p); // Objeto local (const. parámetros)  
        OtraClase obj3; // Objeto local (const. sin parámetros)  
        ....  
    }  
};
```

Ejemplo. La ecuación de una recta viene dada en la forma

$$Ax + By + C = 0$$

es decir, todos los puntos (x, y) que satisfacen dicha ecuación forman una recta.

Queremos definir la clase `Recta` con los métodos básicos y trabajar con un método muy específico que nos indique si una recta pasa por el origen de coordenadas. Usaremos también la clase `Punto2D`.

Cabecera:

Punto2D
- double abscisa
- double ordenada
+ Punto2D (double la_abscisa, double la_ordenada)
+ void SetCoordenadas (double la_abscisa, double la_ordenada)
+ double Abscisa()
+ double Ordenada()

Recta
- double A
- double B
- double C
+ Recta (double val_A, double val_B, double val_C)
+ void SetCoeficientes (double val_A, double val_B, double val_C)
+ double GetA()
+ double GetB()
+ double GetC()
+ double Ordenada (double x)
+ double Abscisa (double y)
+ double Pendiente()
+ bool PasaPorOrigenCoordenadas()

Llamada:

```
int main(){  
  
    .....  
    if (una_recta.PasaPorOrigenCoordenadas())  
        cout << "La recta pasa por el origen de coordenadas";  
    else  
        cout << "La recta no pasa por el origen de coordenadas";
```

Implementación:

```
// Función global para comparar dos datos double (conocida)  
bool SonIguales (double uno, double otro) {  
    return (abs(uno-otro) <= UMBRAL_IGUALES);  
}  
-----  
  
class Punto2D {  
    .....  
};  
  
class Recta {  
    .....  
    bool PasaPorOrigenCoordenadas() {  
  
        Punto2D origen (0,0); // Objeto local (const. parámetros)  
  
        double ordenada_recta = Ordenada(origen.Abscisa());  
        return SonIguales(ordenada_recta, origen.Ordenada());  
    }  
    .....  
};
```

VI.1.3. Pasando objetos como parámetros a los métodos

Un método puede recibir como parámetro un objeto de OTRA clase

```
class UnaClase {  
    ....  
    UnaClase (int dato1, int dato2) {.....}  
    ....  
};  
class OtraClase {  
    ....  
    OtraClase () {.....}  
    void UnMetodoOtraClase (UnaClase objeto_UnaClase){  
        ....  
};  
  
int main(){  
  
    UnaClase un_objeto (6,7);  
    OtraClase otro_objeto;  
  
    otro_objeto.UnMetodoOtraClase (un_objeto);  
    ....
```

En la llamada

```
otro_objeto.Un Metodo Otra Clase (un_objeto);
```

El parámetro real `un_objeto` se empareja con el parámetro formal `objeto_UnaClase`. Recuerde la cabecera del método llamado:

```
void Un Metodo Otra Clase (UnaClase objeto_UnaClase);
```

El emparejamiento consiste en que `objeto_UnaClase` recibe una copia de `un_objeto`. Esta copia se realiza a través del **constructor de copia** ya que el parámetro formal se crea en el momento de la llamada al método.

Al pasar los objetos como parámetros, se invoca automáticamente al constructor de copia

Ejemplo.

Retomamos el ejemplo presentado en la página 611. En ese ejemplo se determinaba si una recta pasaba por el origen de coordenadas con un método específico (`PasaPorOrigenCoordenadas`) que comprobaba si el punto (0, 0) (dato local en el método) satisfacía la ecuación de la recta.

Podemos reescribir el método anterior en base a otro más general, el método `Contiene`, que comprueba si un punto cualquiera forma parte de una recta.

Cabecera:

Punto2D
- double abscisa
- double ordenada
+ Punto2D (double la_abscisa, double la_ordenada)
+ void SetCoordenadas (double la_abscisa, double la_ordenada)
+ double Abscisa()
+ double Ordenada()

Recta
- double A
- double B
- double C
+ Recta (double val_A, double val_B, double val_C)
+ void SetCoeficientes (double val_A, double val_B, double val_C)
+ double GetA()
+ double GetB()
+ double GetC()
+ double Ordenada (double x)
+ double Abscisa (double y)
+ double Pendiente()
+ bool PasaPorOrigenCoordenadas()
+ bool Contiene (Punto2D punto)

Llamada:

```
int main(){
    .....
    if (una_recta.Contiene(un_punto))
        cout << "La recta contiene el punto";
    else
        cout << "La recta no contiene el punto";
```

Implementación:

```
class Recta {
    .....
    bool Contiene (Punto2D punto) {

        double ordenada_recta = Ordenada(punto.Abscisa());
        return SonIguales(ordenada_recta , punto.Ordenada());

        /* También sería correcto:
           return SonIguales(A * punto.Abscisa() +
                               B * punto.Ordenada() + C, 0.0);
        */
    }

    bool PasaPorOrigenCoordenadas() {

        Punto2D origen (0,0); // Objeto local (const. parámetros)
        return (Contiene (origen));
    }
    .....
};
```

Ejemplo. Sobre la clase SecuenciaCaracteres, añadir un método para borrar un conjunto de posiciones dadas por una secuencia de enteros.

```
secuencia --> {'a', 'b', 'c', 'd', 'e', 'f'}
posiciones_a_borrar --> {1, 3}
secuencia --> {'a', 'c', 'e', 'f'}
```

Cabecera:

SecuenciaCaracteres
.....
.....
+ void EliminaVarios (SecuenciaEnteros a_borrar)

Llamada:

```
int main(){

    SecuenciaCaracteres secuencia;
    SecuenciaEnteros posiciones_a_borrar;

    secuencia.Anade('a'); secuencia.Anade('b'); secuencia.Anade('c');
    secuencia.Anade('d'); secuencia.Anade('e'); secuencia.Anade('f');

    posiciones_a_borrar.Anade(1);
    posiciones_a_borrar.Anade(3);

    secuencia.EliminaVarios (posiciones_a_borrar);
```

Implementación:

La implementación de este método lo dejamos como ejercicio.

Un método puede recibir como parámetro un objeto de la MISMA clase

Ejemplo. Queremos añadir un conjunto de caracteres a una secuencia de la clase SecuenciaCaracteres.

```
secuencia --> {'a', 'b', 'c', 'd', 'e', 'f'}
caracteres_a_aniadir --> {'g', 'h'}
secuencia --> {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'}
```

Cabecera:

SecuenciaCaracteres
.....
.....
+ void AniadeVarios (SecuenciaCaracteres nuevos)

Llamada:

```
int main(){

    SecuenciaCaracteres secuencia;
    SecuenciaCaracteres caracteres_a_aniadir;

    secuencia.Aniaide('a'); secuencia.Aniaide('b'); secuencia.Aniaide('c');
    secuencia.Aniaide('d'); secuencia.Aniaide('e'); secuencia.Aniaide('f');

    caracteres_a_aniadir.Aniaide('g');
    caracteres_a_aniadir.Aniaide('h');

    secuencia.AniadeVarios (caracteres_a_aniadir);
    .....
```

Implementación:

```
class SecuenciaCaracteres {  
    ....  
    void AniadeVarios (SecuenciaCaracteres nuevos) {  
  
        int totales_a_aniadir = nuevos.TotalUtilizados();  
  
        for (int i = 0; i < totales_a_aniadir; i++)  
            Aniade(nuevos.Elemento(i));           // Es importante entender  
    }                                         // esta linea  
    ....  
};
```

Los objetos, al igual que los datos de tipos simples, se pueden pasar como parámetros. Es más, los métodos de una clase podrán recibir como parámetros objetos de esa misma clase.

VI.1.4. Métodos que devuelven objetos

Hemos visto en la sección VI.1.3 que un método puede recibir como parámetro un objeto. Nos planteamos ahora si un método puede devolver un objeto (de la misma clase o de otra). La respuesta es **sí**.

La manera habitual de hacerlo será crear un objeto local de la clase a devolver, trabajar sobre él y devolverlo.

- ▷ El objeto local se creará en la pila, dentro del marco del método que se está ejecutando. Para crear el objeto local actuará el *constructor* que se ajuste a la declaración.
- ▷ El objeto existirá mientras se esté ejecutando el método.
- ▷ Al devolver (instrucción `return`) el objeto se creará una copia del mismo (*constructor de copia*) y se libera la memoria asociada al objeto local.

En la función `main` (o desde dónde se llame al método) podremos utilizar el objeto devuelto de dos formas:

1. A través del operador de asignación.
2. A través del constructor de copia.

Nota. Al ejecutar la sentencia `return` también se produce una llamada al constructor de copia, ya que es necesario crear un objeto temporal con los datos del objeto local, para así poder realizar la devolución del objeto.

Un método de una clase puede devolver un objeto de OTRA clase

```
class UnaClase {  
    ....  
};  
  
class OtraClase {  
    ....  
    UnaClase Metodo() {  
        UnaClase objeto_local; // Crear objeto local  
        .... // Procesar objeto local  
        return objeto_local; // Copia del objeto local  
    }  
};  
  
int main(){  
  
    UnaClase receptor1; // Const. sin argumentos  
    OtraClase trabajador; // Const. sin argumentos  
  
    receptor1 = trabajador.Metodo(); // Op. de asignación  
  
    UnaClase receptor2 (trabajador.Metodo()); // Const. de copia  
  
    // La siguiente instrucción ejecuta el método ToString() sobre  
    // un objeto de la clase UnaClase. Se trata de un objeto anónimo  
    // temporal cuyo valor se obtiene como copia del valor devuelto  
    // por la ejecución de trabajador.Metodo()  
  
    cout << (trabajador.Metodo().ToString()); // Const. de copia
```

Ejemplo. Añadimos un método a la clase Recta para que obtenga el SegmentoDirigido incluido en la recta delimitado por dos abscisas.

Cabecera:

Recta
.....
+ SegmentoDirigido Segmento(double abs_origen, double abs_final)

Remitimos a la definición de la clase SegmentoDirigido dada en la página 488 (sección V.4.2). En esa definición NO hay un constructor sin argumentos por lo que este código provoca un error de compilación:

Llamada:

```
int main(){
    Recta una_recta (2, 3, -11); // 2x + 3y - 11 = 0

    SegmentoDirigido segmento; // Error de compilación:
    segmento = una_recta.Segmento (3.5, 4.6);
```

En este ejemplo no podremos emplear el operador de asignación para recoger el segmento devuelto por el método Segmento de la clase Recta (necesitamos un objeto existente). Tendremos que recurrir a construir el segmento a partir de la recta, y “asignarlo” con el constructor de copia:

```
int main(){
    Recta una_recta (2, 3, -11); // 2x + 3y - 11 = 0

    // Crear un objeto de la clase SegmentoDirigido con el const. copia
    // que usa el valor devuelto por el método Segmento de la clase Recta
    SegmentoDirigido segmento (una_recta.Segmento (3.5, 4.6));

    cout << "Longitud segmento = " << segmento.Longitud() << endl;
```

Implementación:

```
.....
class Recta {
    .....
    SegmentoDirigido Segmento (double abs_origen, double abs_final) {

        double ord_origen = Ordenada (abs_origen);
        double ord_final = Ordenada (abs_final);

        SegmentoDirigido segmento (abs_origen, ord_origen,
                                   abs_final, ord_final);

        return segmento;

        // Las dos últimas instrucciones podrían resumirse para devolver
        // una copia de un objeto anónimo recién creado:
        // return SegmentoDirigido (abs_origen, ord_origen,
        //                           abs_final, ord_final);
    }
    .....
};
```

Ejemplo. Añadimos un método a la clase SecuenciaCaracteres que devuelva una secuencia de enteros con las posiciones en las que se encuentra un carácter dado. En la función main, llame al método EliminaVarios para eliminar todas las ocurrencias del carácter.

Cabecera:

SecuenciaCaracteres
.....
+ SecuenciaEnteros PosicionesDe (char a_buscar)

Llamada:

```
int main(){

    SecuenciaCaracteres secuencia;      // Const. sin argumentos
    SecuenciaEnteros pos_encontrado;    // Const. sin argumentos
    char buscado;

    .....
    pos_encontrado = secuencia.PosicionesDe(buscado);
    secuencia.EliminaVarios(pos_encontrado);
```

También podríamos haber usado el constructor de copia:

```
int main(){

    SecuenciaCaracteres secuencia;      // Const. sin argumentos
    char buscado;

    .....
    SecuenciaEnteros pos_encontrado(secuencia.PosicionesDe(buscado));
    secuencia.EliminaVarios(pos_encontrado);
```

Implementación:

```
class SecuenciaCaracteres {  
    ....  
    SecuenciaEnteros PosicionesDe (char a_buscar) {  
  
        SecuenciaEnteros posiciones; // Secuencia vacía  
  
        // a_buscar puede estar repetido en la secuencia por lo que  
        // debemos registrar todas las posiciones en las que está  
  
        for (int i = 0; i < total_utilizados; i++){  
            if (vector_privado[i] == a_buscar)  
                posiciones.Anade(i);  
        }  
  
        return posiciones;  
    }  
    ....  
};
```

Ejemplo. Construimos una clase para la lectura de secuencias de caracteres. Este sería un ejemplo de una clase cuya responsabilidad es *fabricar objetos de otra clase*.

Cabecera:

LectorSecuenciaCaracteres
.....
+ SecuenciaCaracteres Lee ()

Llamada:

```
int main() {  
  
    const char TERMINADOR = '#';  
    LectorSecuenciaCaracteres lector_secuencias (TERMINADOR);  
  
    SecuenciaCaracteres secuencia;  
    SecuenciaCaracteres otra_secuencia;  
  
    secuencia = lector_secuencias.Lee();  
    otra_secuencia = lector_secuencias.Lee();  
    ....
```

Implementación:

```
class LectorSecuenciaCaracteres {  
  
private:  
    char terminador;  
  
public:  
    LectorSecuenciaCaracteres (char caracter_terminador)  
        : terminador (caracter_terminador) {}
```

```
SecuenciaCaracteres Lee() {  
  
    SecuenciaCaracteres a_leer;  
  
    int total_introducidos, capacidad_secuencia;  
    char caracter;  
  
    total_introducidos = 0;  
    capacidad_secuencia = a_leer.Capacidad();  
    caracter = cin.get();  
  
    while (caracter == '\n') // Se salta los new line  
        caracter = cin.get();  
  
    while (caracter != terminador &&  
           total_introducidos < capacidad_secuencia){  
  
        a_leer.Anade(caracter);  
        total_introducidos++;  
        caracter = cin.get();  
    }  
  
    return a_leer;  
}  
.....  
};
```

<http://decsai.ugr.es/~carlos/FP/SecuenciaCaracteres.cpp>

Un método de una clase puede devolver un objeto de la MISMA clase

```
class MiClase {  
    ....  
    MiClase Metodo() {  
  
        MiClase objeto;  
        ....  
        return objeto;  
    }  
    ....
```

Ejemplo. Sobre la clase SecuenciaCaracteres, añadimos dos métodos que devuelven una *nueva secuencia de caracteres*, con los caracteres de la secuencia transformados a mayúsculas y minúsculas, respectivamente.

Cabecera:

SecuenciaCaracteres
.....
+ SecuenciaCaracteres ToUpper()
+ SecuenciaCaracteres ToLower()

Llamada:

```
int main(){  
    SecuenciaCaracteres secuencia;  
  
    secuencia.Ania('T');    secuencia.Ania('e');  
    secuencia.Ania('c');    secuencia.Ania('A');  
  
    cout << secuencia.ToUpper().ToString() << endl;  
    cout << secuencia.ToLower().ToString() << endl;
```

Implementación:

```
class SecuenciaCaracteres {  
    ....  
    SecuenciaCaracteres ToUpper() {  
  
        SecuenciaCaracteres en_mayuscula;  
  
        for(int i = 0; i < total_utilizados; i++)  
            en_mayuscula.Anade(toupper(vector_privado[i]));  
  
        return en_mayuscula;  
    }  
  
    SecuenciaCaracteres ToLower() {  
  
        SecuenciaCaracteres en_minuscula;  
  
        for(int i = 0; i < total_utilizados; i++)  
            en_minuscula.Anade(tolower(vector_privado[i]));  
  
        return en_minuscula;  
    }  
    ....  
};
```

Ejemplo. Dada la recta $Ax + By + C = 0$, la ecuación de una recta perpendicular a ésta pasando por el punto (x_0, y_0) es $m'(x - x_0) + (y - y_0) = 0$ (de otra forma: $m'x + y - m'x_0 - y_0 = 0$) donde m' es la pendiente de la nueva recta $m' = B/A$.

Cabecera:

Recta
.....
+ Recta Perpendicular (Punto2D punto_ref)

Llamada:

```
int main(){
    Punto2D un_punto(5, 3);
    Recta una_recta(2, 3, -11);      // 2x + 3y - 11 = 0
    Recta recta_perpendicular (una_recta.Perpendicular (un_punto));
    cout << "Pendiente = " << recta_perpendicular.Pendiente();
```

Implementación:

```
class Recta{
    .....
    Recta Perpendicular (Punto2D punto_ref) {
        double coef_y = 1;
        double pendiente_nueva = B/A;
        double coef_x = -pendiente_nueva;
        double coef_ind = punto_ref.Abscisa() * pendiente_nueva
                        - punto_ref.Ordenada();
        Recta perpendicular(coef_x, coef_y, coef_ind);
        return perpendicular;
    }
};
```

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

Métodos y vectores clásicos

Recordatorio 1: Un método/función no puede recibir/devolver un vector.¹

```
class MiClase{  
    ....  
    int [] Metodo(){           // Error de compilación  
        int vector [TAMANIO];  
        ....  
        return vector;  
    }  
    ....
```

Recordatorio 2: Podemos declarar y usar un vector clásico dentro de un método/función (es un *dato local* al método).

```
class MiClase {  
    ....  
    int Metodo() {  
        int vector_local [TAMANIO];  
        ....  
        return un_valor_int;  
    }  
    ....
```

Recordatorio 3: Un método/función puede devolver un objeto.

Podemos concluir que no hay ningún inconveniente para que un método pudiera recibir y devolver un objeto que contuviera como dato miembro un vector clásico.

¹Realmente no recibe un vector como tal sino la dirección de memoria donde empiezan sus datos -se verá en *Metodología de la Programación-*

VI.1.5. Operaciones binarias entre objetos de una misma clase

¿Cómo diseñamos los métodos que trabajan sobre dos objetos de una misma clase?

Las operaciones binarias que involucren dos objetos de una misma clase, usualmente se implementarán como métodos de dicha clase. El método actuará sobre un objeto y se le pasará como parámetro el otro

IMPORTANT

En estos casos tenemos dos objetos:

- ▷ El objeto sobre el que se ejecuta el método, llamado *objeto implícito*. En el código del método el acceso a los datos y métodos de este objeto se realiza directamente, sin escribir nada más que el nombre del dato o método.
- ▷ El objeto que se recibe como argumento, llamado *objeto explícito*. Este nombre viene del hecho que ese objeto tiene nombre (el indicado en la lista de parámetros) y en consecuencia, el acceso a los datos y métodos del objeto explícito se realiza utilizando el operador punto.

Ejemplos:

```
int pos_contiene = secuencia.PosicionContiene(pequenia);
bool se_intersecan = circunferencia.Interseca(otra_circunferencia);
son_iguales = un_punto.EsIgual_a(otro_punto);
Conjunto cjto_union = un_conjunto.Union(otro_conjunto);
Fraccion suma_fracciones(un_fraccion.Suma(otra_fraccion))
```

Nota:

Usualmente en el nombre de los métodos omitiremos los sufijos "le", "_con", "_a", etc. Por ejemplo: `una_fraccion.Sumar(...)`, `coordenadaGPS.Distancia(otra_coordenada)`, etc.

A veces, está claro cuales son los objetos implícito y explícito. Otras no, y la confusión puede dar lugar a un mal diseño.

Ejemplo. Defina un método para ver si una secuencia de caracteres contiene a otra. El algoritmo lo conocemos. Esta solución proporciona un método para esa tarea, devolviendo la posición de la secuencia *grande* donde empieza la secuencia *pequeña*, ó -1 si no hay coincidencia.

Cabecera:

SecuenciaCaracteres
.....
+ int PosicionContiene (SecuenciaCaracteres pequenia)

Llamada:

```
int main(){
    .....
    pos = secuencia.PosicionContiene (pequenia);

    if (pos == -1)
        cout << "No encontrada";
    else
        cout << "Encontrada en la posición " << pos;
    .....
}
```

Lo que nunca haremos será definir el método Contiene pasándole dos secuencias como parámetros:

```
SecuenciaCaracteres esta_no_pinta_nada;
SecuenciaCaracteres secuencia, pequenia;
.....
pos = esta_no_pinta_nada.Contiene(secuencia, pequenia);
```



Implementación:

```
int PosicionContiene (SecuenciaCaracteres pequenia) {  
  
    int posicion_contiene;  
    bool encontrado;  
    int utilizados_pequenia = pequenia.TotalUtilizados();  
  
    if (utilizados_pequenia > 0) {  
  
        int ultima_pos_posible = total_utilizados - utilizados_pequenia;  
        encontrado = false;  
        int inicio = 0;  
  
        while (inicio <= ultima_pos_posible && !encontrado) {  
            bool sigo = true;  
            int i = 0;  
            while (i < utilizados_pequenia && sigo) {  
                sigo=(vector_privado[inicio+i]==pequenia.Elemento(i));  
                i++;  
            }  
            if (sigo) { // ÉXITO: Se encontró pequenia  
                posicion_contiene = inicio;  
                encontrado = true;  
            }  
            else inicio++; // Empezar otra secuencia  
        }  
    }  
    else encontrado = false; // pequenia está vacía  
  
    return ((encontrado) ? posicion_contiene : -1);  
}
```

Ejemplo. Vamos a darle la vuelta a los argumentos del ejemplo anterior. En lugar de buscar si una secuencia (*la grande*) contiene a otra (*la pequeña*) ahora buscamos si la secuencia *pequeña* está contenida en *la grande*.

Cabecera:

SecuenciaCaracteres
.....
+ int PosicionContenida (SecuenciaCaracteres grande)

Llamada:

```
int main(){  
    .....  
    pos = secuencia.PosicionContenida (grande);  
  
    if (pos == -1)  
        cout << "No encontrada";  
    else  
        cout << "Encontrada en la posición " << pos;  
    .....  
}
```

Implementación:

```
int PosicionContenida (SecuenciaCaracteres grande) {  
  
    int posicion_contenida;  
    bool encontrado;  
    int utilizados_grande = grande.TotalUtilizados();  
  
    if (utilizados_pequenia > 0) {  
  
        int ultima_pos_posible = utilizados_grande - total_utilizados;  
        encontrado = false;  
        int inicio = 0;  
  
        while (inicio <= ultima_pos_posible && !encontrado) {  
            bool sigue = true;  
            int i = 0;  
            while (i < total_utilizados && sigue) {  
                sigue=(vector_privado[i]==grande.Elemento(inicio+i));  
                i++;  
            }  
            if (sigue) { // ÉXITO: Se encontró pequenia  
                posicion_contenida = inicio;  
                encontrado = true;  
            }  
            else inicio++; // Empezar otra secuencia  
        }  
    }  
    else encontrado = false; // pequenia está vacía  
  
    return ((encontrado) ? posicion_contenida : -1);  
}
```

En otras ocasiones los dos objetos intervienen con idéntica *importancia*. En este caso, se fija cualquiera de ellos y se pasa como parámetro el otro.

Ejercicio. Añadimos a Fraccion un método para sumarle otra fracción.

```
Fraccion una_fraccion(4, 9), otra_fraccion(5,2);
Fraccion suma_v1 (una_fraccion.Suma(otra_fraccion));
Fraccion suma_v2 (otra_fraccion.Suma(una_fraccion));
```

Cabecera:

Fraccion	
- int	numerador
- int	denominador
+ Fraccion	(int el Numerador, int el Denominador)
+ void	SetNumerador (int el Numerador)
+ void	SetDenominador (int el Denominador)
+ int	Numerador()
+ int	Denominador()
+ void	Reduce()
+ double	Division()
+ string	ToString()
+ Fraccion	Suma(Fraccion otra_fraccion)

Llamada:

```
int main(){

    int numerador, denominador;

    cout << "Numerador y denominador de la primera fraccion: ";
    cin >> numerador;
    cin >> denominador;

    Fraccion una_fraccion (numerador, denominador);
```

```
cout << endl;
cout << "Numerador y denominador de la segunda fraccion: ";
cin >> numerador;
cin >> denominador;

Fraccion otra_fraccion(numerador, denominador);

Fraccion suma_fracciones (una_fraccion.Suma(otra_fraccion));
suma_fracciones.Reduce();

cout << endl;
cout << "Suma = " << suma_fracciones.ToString() << endl;
```

Implementación:

```
class Fraccion {
    .....
    Fraccion Suma (Fraccion otra_fraccion){
        int suma Numerador;
        int suma Denominador;

        suma Numerador = numerador * otra_fraccion.Denominador() +
                          denominador * otra_fraccion.Numerador();
        suma Denominador = denominador * otra_fraccion.Denominador();

        Fraccion suma (suma Numerador, suma Denominador);
        suma.Reduce();

        return suma;
    }
    .....
};
```

Un error muy común en los exámenes es definir el método y pasar dos objetos como parámetros:

```
Fraccion absurda(3, 5);  
Fraccion una_fraccion(4, 9), otra_fraccion(5,2);  
Fraccion suma (absurda.Suma(una_fraccion, otra_fraccion));
```



Otro error común es definir una clase específica para realizar operaciones con dos fracciones:

```
class ParejaFracciones { 😞  
private:  
    Fraccion una_fraccion;  
    Fraccion otra_fraccion;  
public:  
    ParejaFracciones (Fraccion la_primera, Fraccion la_segunda)  
        : una_fraccion(la_primera), otra_fraccion(la_segunda) {}  
    Fraccion Suma()  
    {  
        int suma Numerador = una_fraccion.Numerador() *  
                        otra_fraccion.Denominador() +  
                        una_fraccion.Denominador() *  
                        otra_fraccion.Numerador();  
        int suma Denominador = una_fraccion.Denominador() *  
                                otra_fraccion.Denominador();  
        Fraccion suma(suma Numerador, suma Denominador);  
        suma.Reduce();  
        return suma;  
    }  
};
```

Si tuviésemos que definir una clase Pareja_C para cada clase C sobre la que quisiésemos realizar operaciones con pares de objetos, el código se complicaría innecesariamente.

Como ya dijimos, tampoco definiremos funciones globales que actúen sobre objetos:

```
#include <iostream>

// Clase
class Fraccion {
    .....
};

// Función
Fraccion Suma (Fraccion una_fraccion, Fraccion otra_fraccion){
    .....
}
```

Ejemplo. Comprobar si dos puntos son iguales.

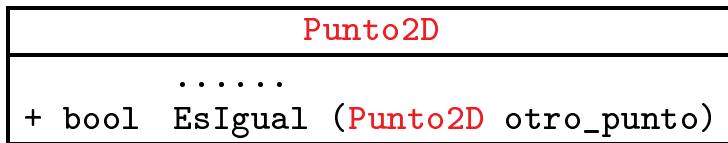
Con una función global sería:

```
bool SonIguales (double un_real, double otro_real){
    return abs(un_real-otro_real) <= UMBRAL;
}

bool SonIgualesPuntos(Punto2D un_punto, Punto2D otro_punto){
    return (SonIguales(un_punto.Abscisa(), otro_punto.Abscisa())
        && SonIguales(un_punto.Ordenada(), otro_punto.Ordenada()));
}
```

Preferimos que se haga con métodos. Observe como hay que adaptar el nombre.

Cabecera:



Llamada:

```
int main(){

    Punto2D un_punto(5.1, 3.2);
    Punto2D otro_punto(5.1, 3.5);

    bool son_iguales = un_punto.EsIgual (otro_punto);
```

Implementación:

```
bool SonIguales (double un_real, double otro_real){  
    return fabs(un_real-otro_real) <= UMBRAL;  
}  
  
class Punto2D{  
    ....  
    bool EsIgual (Punto2D otro_punto){  
        return (SonIguales(abscisa, otro_punto.Abscisa()) &&  
                SonIguales(ordenada, otro_punto.Ordenada()));  
    }  
};
```



VI.1.6. Acceso a los datos miembros privados

Si a un método de una clase le pasamos como parámetro (o declaramos local) un objeto *de la misma clase*, podremos acceder a sus datos miembros *privados* usando la notación con punto sin necesidad de emplear los métodos de consulta.

Es coherente que ésto se permita con objetos de la misma clase. De hecho, la mayor parte de los lenguajes orientados a objetos lo permiten.

Ejemplo. Implementamos el método `EsIgual` de la clase `Punto2D` de otra forma alternativa:

Antes:

```
class Punto2D{  
    ....  
    bool EsIgual (Punto2D otro_punto){  
        return (SonIguales(abscisa, otro_punto.Abscisa()) &&  
                SonIguales(ordenada, otro_punto.Ordenada()));  
    }  
};
```

Ahora:

```
class Punto2D{  
    ....  
    bool EsIgual (Punto2D otro_punto){  
        return (SonIguales(abscisa, otro_punto.abscisa) &&  
                SonIguales(ordenada, otro_punto.ordenada));  
    }  
};
```

Ejemplo. Implementamos el método Suma de la clase Fraccion de otra forma alternativa.

Antes:

```
class Fraccion{  
    ....  
    Fraccion Suma (Fraccion otra_fraccion) {  
        int suma_numerator = numerator * otra_fraccion.Denominador() +  
                            denominador * otra_fraccion.Numerador();  
        int suma_denominador = denominador * otra_fraccion.Denominador();  
  
        Fraccion suma (suma_numerator, suma_denominador);  
        suma.Reduce();  
  
        return suma;  
    }  
    ....
```

Ahora:

```
class Fraccion{  
    ....  
    Fraccion Suma (Fraccion otra_fraccion) {  
        int suma_numerator = numerator * otra_fraccion.denominador +  
                            denominador * otra_fraccion.numerador;  
        int suma_denominador = denominador * otra_fraccion.denominador;  
  
        Fraccion suma (suma_numerator, suma_denominador);  
        suma.Reduce();  
  
        return suma;  
    }  
    ....
```

Ejemplo. Implementamos el método AniadeVarios de la clase SecuenciaCaracteres de otra forma alternativa.

Antes:

```
class SecuenciaCaracteres {  
    ....  
    void AniadeVarios (SecuenciaCaracteres nuevos) {  
  
        int totales_a_aniadir = nuevos.TotalUtilizados();  
  
        for (int i = 0; i < totales_a_aniadir; i++)  
            Aniade (nuevos.Elemento(i));  
    }  
    ....  
};
```

Ahora:

```
class SecuenciaCaracteres{  
    ....  
    void AniadeVarios (SecuenciaCaracteres nuevos) {  
  
        int totales_a_aniadir = nuevos.TotalUtilizados();  
  
        for (int i = 0; i < totales_a_aniadir; i++)  
            Aniade (nuevos.vector_privado[i]);  
    }  
    ....  
};
```

Si tenemos un objeto local, podemos modificar sus datos miembros privados, aunque usualmente lo haremos a través de los métodos

```
class SecuenciaCaracteres {  
    ....  
    SecuenciaCaracteres ToUpper() { 😊  
        SecuenciaCaracteres en_mayuscula;  
        for (int i = 0; i < total_utilizados; i++)  
            en_mayuscula.Anade (toupper(vector_privado[i]));  
        return en_mayuscula;  
    }  
};  
class SecuenciaCaracteres{  
    ....  
    SecuenciaCaracteres ToUpper() { 😞  
        SecuenciaCaracteres en_mayuscula;  
  
        for (int i = 0; i < total_utilizados; i++) {  
            en_mayuscula.vector_privado[i] = toupper(vector_privado[i]);  
            en_mayuscula.total_utilizados++; // ;¡Que no se olvide!!  
        }  
        return en_mayuscula;  
    }  
};
```

Es preferible la primera versión ya que la tarea de añadir (asignar e incrementar el total de utilizados) es responsabilidad del método Anade.

Puede acceder a los datos miembros privados de otros objetos de la misma clase, pero reserve esta posibilidad para los casos en los que sólo necesite consultar su valor.

VI.2. Objetos como datos miembro de otros objetos

¿Un objeto puede ser dato miembro de otro objeto? Por supuesto.

- ▷ Como cualquier otro dato miembro, C++ permite que sea privado o público. Nosotros siempre usaremos datos privados.
- ▷ Su uso sigue las mismas normas de programación que cualquier otro dato miembro. Sólo usaremos objetos como datos miembro, cuando esté claro que forman parte del *núcleo* de la clase.
- ▷ La clase del objeto contenido debe declararse antes que la clase del objeto contenedor.
- ▷ La clase contenedora y la clase contenida no pueden ser la misma (con punteros sí se puede -Metodología de la Programación-)
- ▷ En el objeto de la clase contenedora se crea una instancia de la clase contenida.

```
class ClaseContenida {  
    <datos miembro y métodos>  
};  
  
class ClaseContenedora {  
private:  
    ClaseContenida dato_miembro;  
};
```

¿Qué ocurre cuando el constructor de la clase contenida tiene parámetros? Hay que crear el objeto con los parámetros adecuados. ¿Dónde? **Obligatoriamente, en la lista de inicialización del constructor.**

```
class ClaseContenida {  
public:  
    ClaseContenida (int parametro){  
        .....  
    }  
    .....  
};  
class ClaseContenedora {  
private:  
    // ClaseContenida dato_miembro(5);  Error compilación  
    ClaseContenida dato_miembro;  
public:  
    ClaseContenedora() : dato_miembro(5)  
    // Llamada al constructor con parámetros de ClaseContenida  
    { ..... }  
    .....  
};
```

O en general:

```
class ClaseContenedora {  
private:  
    ClaseContenida dato_miembro;  
public:  
    ClaseContenedora (int valor) : dato_miembro(valor)  
    // Llamada al constructor con parámetros de ClaseContenida  
    { ..... }  
    .....  
};
```

Ejemplo. Segmento.

Dada la clase Punto2D:

Punto2D
- double abscisa
- double ordenada
+ Punto2D (double abs_punto, double ord_punto)
+ void SetCoordenadas (double abs_punto, double ord_punto)
+ double GetAbscisa()
+ double GetOrdenada()
+ bool EsIgual (Punto2D otro_punto)

Redefinimos la clase SegmentoDirigido para que contenga dos puntos como datos miembro en vez de cuatro double.

Al constructor del segmento le pasamos cuatro double (las coordenadas) para poder construir los puntos.

SegmentoDirigido
- Punto2D origen
- Punto2D final
+ SegmentoDirigido (double abs_origen, double ord_origen, double abs_final, double ord_final)
+ void SetCoordenadas (double abs_origen, double ord_origen, double abs_final, double ord_final)
+ Punto2D GetOrigen()
+ Punto2D GetFinal()
+ double Longitud()
+ void TrasladaHorizontal (double unidades)
+ void TrasladaVertical (double unidades)
+ void Traslada (double en_horizontal, double en_vertical)

```
class SegmentoDirigido {  
  
private:  
    Punto2D origen;  
    Punto2D final;  
  
public:  
  
    // Constructor SegmentoDirigido con parámetros de tipo double  
    SegmentoDirigido(double abs_origen, double ord_origen,  
                      double abs_final, double ord_final)  
        // Llamada al constructor de los puntos -con parámetros-  
        : origen (abs_origen, ord_origen),  
          final (abs_final, ord_final)  
    {}  
  
    Punto2D GetOrigen () {  
        return origen;  
    }  
    Punto2D GetFinal () {  
        return final;  
    }  
    double Longitud () {  
        double dif_abscisa = origen.GetAbscisa()-final.GetAbscisa();  
        double dif_ordenada = origen.GetOrdenada()-final.GetOrdenada();  
        return sqrt(dif_abscisa*dif_abscisa + dif_ordenada*dif_ordenada);  
    }  
    void TrasladaHorizontal (double unidades) {  
        origen.SetCoordenadas(origen.GetAbscisa() + unidades,  
                              origen.GetOrdenada());  
        final.SetCoordenadas (final.GetAbscisa() + unidades,  
                             final.GetOrdenada());  
    }  
}
```

```
void TrasladaVertical(double unidades) {  
    origen.SetCoordenadas(origen.GetAbscisa(),  
                          origen.GetOrdenada() + unidades);  
    final.SetCoordenadas (final.GetAbscisa(),  
                          final.GetOrdenada() + unidades);  
}  
void Traslada (double en_horizontal, double en_vertical) {  
    TrasladaHorizontal(en_horizontal);  
    TrasladaVertical(en_vertical);  
}  
};  
  
int main(){  
  
    SegmentoDirigido segmento (3.4, 4.5, 6.7, 9.2);  
  
    cout << segmento.GetOrigen().GetAbscisa(); // 3.4  
    cout << segmento.GetOrigen().GetOrdenada(); // 4.5  
  
    Punto2D copia_de_origen (segmento.GetOrigen());  
  
    cout << copia_de_origen.GetAbscisa(); // 3.4  
    cout << copia_de_origen.GetOrdenada(); // 4.5
```

¿Qué ocurre si cambiamos las coordenadas de copia_de_origen?

```
copia_de_origen.SetCoordenadas (9.1, 10.2);
```

Obviamente, no se modifican las coordenadas del origen del segmento.

El anterior ejemplo pone de manifiesto que:

A lo largo de esta asignatura, sólo se trabaja con copias de objetos. En particular, los métodos siempre devuelven copias de objetos.

Por lo tanto, si un objeto *A* tiene como dato miembro un objeto *B* y un método de *A* devuelve una copia de *B*, las modificaciones que se hagan sobre dicha copia no afectan a *B*.

IMPORTANT

En vez de construir el objeto *interno* en la clase contenedora, también podríamos construir el objeto fuera y pasarlo como parámetro al constructor. En dicho caso, debemos inicializar el dato miembro en la lista de inicialización, asignándole una copia del objeto pasado como parámetro (el compilador invoca al constructor de copia del dato miembro)

```
class ClaseContenida {  
public:  
    ClaseContenida (int parametro){  
        .....  
    }  
    <datos miembro y métodos>  
};  
  
class ClaseContenedora {  
  
private:  
    ClaseContenida dato_miembro;  
  
public:  
    ClaseContenedora(ClaseContenida objeto)  
        : dato_miembro(objeto)  
        // Llamada al constructor de copia de ClaseContenida  
    { ..... }  
    .....  
};
```

Ejemplo. Segmento. Constructor con objetos como parámetros.

SegmentoDirigido
- Punto2D origen
- Punto2D final
+ SegmentoDirigido (Punto2D punto_origen, Punto2D punto_final)
+ void SetCoordenadas (Punto2D punto_origen, Punto2D punto_final)
+ Punto2D GetOrigen()
+ Punto2D GetFinal()
+ double Longitud()
+ void TrasladaHorizontal(double unidades)
+ void TrasladaVertical(double unidades)
+ void Traslada(double en_horizontal, double en_vertical)

Al constructor del segmento le pasamos directamente dos objetos de la clase Punto2D, en vez de los cuatro double de las coordenadas.

```
class SegmentoDirigido {  
  
private:  
    Punto2D origen;  
    Punto2D final;  
  
public:  
  
    // Constructor SegmentoDirigido con parámetros de tipo Punto2D  
    SegmentoDirigido (Punto2D punto_origen, Punto2D punto_final)  
        : origen (punto_origen), // llamada al constructor de copia del punto  
          final (punto_final) // llamada al constructor de copia del punto  
    {}  
    ....  
};
```

```
int main() {  
  
    Punto2D origen(3.4, 4.5);  
    Punto2D final(6.7, 9.2);  
  
    SegmentoDirigido segmento (origen, final);  
  
    cout << segmento.GetOrigen().GetAbscisa(); // 3.4  
    cout << segmento.GetOrigen().GetOrdenada(); // 4.5
```

También es válido utilizar *objetos anónimos*:

```
int main() {  
  
    SegmentoDirigido segmento (Punto2D (3.4,4.5), Punto2D (6.7,9.2));  
  
    cout << segmento.GetOrigen().GetAbscisa(); // 3.4  
    cout << segmento.GetOrigen().GetOrdenada(); // 4.5
```

Ejemplo. Si usamos esta versión del segmento, el método Segmento de la clase Recta visto en la página 623 que devolvía el segmento delimitado por dos valores de abscisas habría que reescribirlo como sigue:

```
class Punto2D {  
    ....  
};  
  
class Recta {  
private:  
    ....  
public:  
    ....  
    SegmentoDirigido Segmento(double abs_origen, double abs_final){  
  
        double ord_origen = GetOrdenada(abs_origen);  
        double ord_final = GetOrdenada(abs_final);  
  
        Punto2D pto_origen(abs_origen, ord_origen);  
        Punto2D pto_final(abs_final, ord_final);  
  
        SegmentoDirigido segmento(pto_origen, pto_final);  
  
        return segmento;  
    }  
};
```

Ejemplo. Añadimos a la clase CuentaBancaria un dato miembro de tipo Fecha para representar la fecha de apertura.

CuentaBancaria
- double saldo
- double identificador
- Fecha fecha_apertura
- void SetSaldo(double saldo_propuesto)
- void SetIdentificador(string identificador_cuenta)
+ CuentaBancaria(string identificador_cuenta, double saldo_inicial, Fecha fecha_apertura_cuenta)
+ string Identificador()
+ double Saldo()
.....

```
class CuentaBancaria{  
private:  
    Fecha fecha_apertura;  
    double saldo;  
    const string identificador;  
public:  
    CuentaBancaria(string identificador_cuenta, double saldo_inicial,  
                    Fecha fecha_apertura_cuenta)  
        : saldo (saldo_inicial),  
          identificador (identificador_cuenta),  
          fecha_apertura (fecha_apertura_cuenta)  
        // Constructor de copia de Fecha  
    {}  
    .....  
};  
  
int main(){  
    Fecha una_fecha(12, 11, 2015);  
    CuentaBancaria cuenta("2031450100001367", 2000, una_fecha);  
    .....  
}
```

Si se quiere inicializar un objeto dato miembro a unos valores por defecto, se puede hacer de dos formas:

- ▷ **Usando la inicialización de los datos miembros entre llaves (disponible en C++ 11):**

```
class SegmentoDirigido {  
private:  
    Punto2D origen = {0.0, 0.0};  
    Punto2D final = {1.0, 1.0};  
    ....
```

- ▷ **Recomendado: Usando en la lista de inicialización del constructor un *objeto anónimo* :**

```
class SegmentoDirigido {  
private:  
    Punto2D origen;  
    Punto2D final;  
public:  
    SegmentoDirigido()  
        : origen (Punto2D(0.0, 0.0)),  
          final (Punto2D(1.0, 1.0))  
    { }  
    ....
```

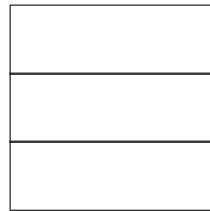
VI.3. Tablas de datos

De forma genérica, diremos que una tabla es un conjunto de filas.

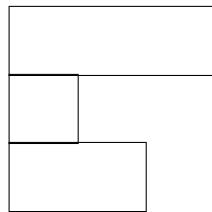
Las tablas serán para las matrices lo que las secuencias para los vectores: nos permite encapsular una estructura de datos (una *matriz clásica* en este caso) en un objeto y gestionarla mediante un conjunto de métodos públicos.

Distinguiremos las siguientes situaciones:

- ▷ **Tabla rectangular:** todas las filas tienen el mismo número de columnas.



- ▷ **Tabla dentada:** el número de columnas en cada fila no tiene por qué ser el mismo.



Veamos cómo representar este tipo de estructuras:

- ▷ Usando matrices clásicas.
- ▷ Usando vectores de objetos.

VI.3.1. Representación de una tabla con matrices

La representación basada en una matriz clásica quiere decir que la estructura de datos que se empleará para almacenar los datos será una *matriz clásica*, declarada en la zona privada de la clase.

VI.3.1.1. Tabla rectangular usando una matriz

X	X	X	?	?	?	...	?
X	X	X	?	?	?	...	?
?	?	?	?	?	?	...	?
...
?	?	?	?	?	?	...	?

Si se trata de una *tabla rectangular* *todas las filas tienen el mismo número de columnas ocupadas*. En el caso de una tabla rectangular de caracteres su definición en una clase podría ser:

```

static const int NUM_FILS = 50; // Filas disponibles
static const int NUM_COLS = 40; // Columnas disponibles

char matriz_privada[NUM_FILS] [NUM_COLS];

// PRE: 0 <= filas_utilizadas <= NUM_FILS
// PRE: 0 <= cols_utilizadas <= NUM_COLS

int filas_utilizadas;
int cols_utilizadas;
```

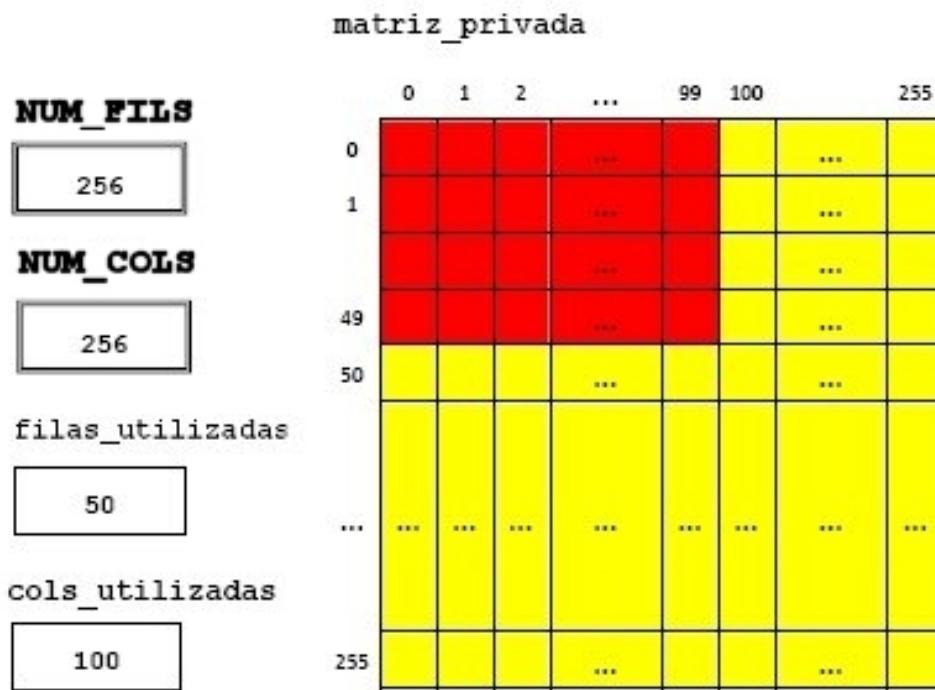


Figura VI.1: Tabla rectangular en la que se ocupan 50×100 casillas de las 256×256 disponibles

1. Todas las filas tendrán el mismo número de columnas. ¿Cómo lo imponemos?

a) Pasando al constructor un entero (número de columnas).

Impondremos como precondition que el número de columnas especificado en el constructor está en el rango correcto.

```
// Recibe "numero_de_columnas" que indica el número de
// columnas que deben tener TODAS las filas.
// PRE: numero_de_columnas <= NUM_COLS
```

```
TablaRectangularCaracteres (int numero_de_columnas)
    : filas_utilizadas(0),
      cols_utilizadas(numero_de_columnas)
{ }
```

b) Pasando al constructor la primera fila.

Impondremos como precondition que el número de casillas de la primera fila está en el rango correcto.

```
// Recibe "primera_fila" (una secuencia de caracteres).
// El número de elementos de la secuencia es el valor que se
// usará como "cols_utilizadas"
// PRE: primera_fila.TotalUtilizados() <= NUM_COLS
```

```
TablaRectangularCaracteres (SecuenciaCaracteres primera_fila)
    : filas_utilizadas(0),
      cols_utilizadas (primera_fila.TotalUtilizados())
{
    Aniade(primera_fila); // Actualiza "filas_utilizadas"
}
```

2. Si vamos a ocupar un bloque rectangular completo (sin huecos), lo lógico es *no permitir añadir caracteres uno a uno sino una fila completa* (que debe tener el número de casillas correcto).

```
// Añade una fila completa (un objeto "SecuenciaCaracteres").
// PRE: fila.TotalUtilizados() = cols_utilizadas
// PRE: filas_utilizadas < NUM_FILS
void Aniade (SecuenciaCaracteres fila)
{
    int numero_columnas_fila = fila.TotalUtilizados();

    if ((filas_utilizadas < NUM_FILS) &&
        (numero_columnas_fila == cols_utilizadas)) {
        for (int col = 0; col < numero_columnas_fila; col++)
            matriz_privada[filas_utilizadas] [col]=fila.Elemento(col);
        filas_utilizadas++;
    }
}
```

3. Podremos recuperar: a) el carácter de una casilla concreta (dentro de la zona ocupada):

```
// Devuelve el carácter de la casilla (fila, columna)
// PRE: 0 <= fila < filas_utilizadas
// PRE: 0 <= columna < cols_utilizadas
```

```
char Elemento (int fila, int columna)
{
    return (matriz_privada[fila] [columna]);
}
```

o b) una fila entera:

```
// Devuelve una fila completa (un objeto "SecuenciaCaracteres")
// PRE: 0 <= indice_fila < filas_utilizadas
```

```
SecuenciaCaracteres Fila (int indice_fila)
{
    SecuenciaCaracteres fila;

    for (int col=0; col<cols_utilizadas; col++)
        fila.Anade(matriz_privada[indice_fila] [col]);

    return (fila);
}
```

4. Ofreceremos también los métodos de consulta habituales:

```
int CapacidadFilas (void)
.....
int CapacidadColumnas (void)
.....
int FilasUtilizadas (void)
.....
int ColumnasUtilizadas (void)
```

A modo de resumen, la clase TablaRectangularCaracteres podría especificarse de acuerdo a esta descripción:

TablaRectangularCaracteres	
- const int	NUM_FILS
- const int	NUM_COLS
- char	matriz_privada[NUM_FILS] [NUM_COLS]
- int	filas_utilizadas
- int	cols_utilizadas
+	TablaRectangularCaracteres(void)
+	TablaRectangularCaracteres(int num_columnas)
+	TablaRectangularCaracteres(SecuenciaCaracteres primera_fila)
+ int	CapacidadFilas(void)
+ int	CapacidadColumnas(void)
+ int	FilasUtilizadas(void)
+ int	ColumnasUtilizadas(void)
+ char	Elemento(int fila, int columna)
+ SecuenciaCaracteres	Fila(int indice_fila)
+ SecuenciaCaracteres	Columna(int indice_fila)
+ void	Aniade(SecuenciaCaracteres fila_nueva)
+ void	Inserta (int num_fila, SecuenciaCaracteres fila_nueva)
+ void	Elimina (int num_fila)
+ void	EliminaTodos (void)
+ bool	EstaVacia (void)
+ bool	EsIgual (TablaRectangularCaracteres otra)

Ejemplo. Sopa de letras

```

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   ?   ...   ?
?   ?   ?   ?   ?   ?   ?   ...   ?
...
?   ?   ?   ?   ?   ?   ?   ...   ?

```

En este ejemplo concreto estamos interesados en buscar la posición de una componente, para lo que resulta útil devolverla usando un struct:

```

struct ParFilaColumna{
    int fila;
    int columna;
};

```

SopaLetras	
- const int	MAX_FIL
- const int	MAX_COL
- const int	util_col
- char	matriz_privada[MAX_FIL] [MAX_COL]
- int	util_fil
+	SopaLetras(int numero_de_columnas)
+	SopaLetras(SecuenciaCaracteres primera_fila)
+ int	CapacidadFilas()
+ int	FilasUtilizadas()
+ int	ColUtilizadas()
+ char	Elemento(int fila, int columna)
+ SecuenciaCaracteres	Fila(int indice_fila)
+ void	Aniade(SecuenciaCaracteres fila_nueva)
+ ParFilaColumna	BuscaPalabra (SecuenciaCaracteres a_buscar)

```
int main(){
    const char TERMINADOR = '#';
    char letra;
    int numero_columnas, longitud_a_buscar;
    ParFilaColumna encontrado;
    SecuenciaCaracteres cadena, a_buscar;

    // Formato entrada: número columnas y caracteres de la sopa por filas
    // El número de caracteres debe ser múltiplo del número de columnas

    cin >> numero_columnas;
    SopaLetras sopa(numero_columnas);

    cin >> letra;
    while (letra != TERMINADOR) {

        for (int i=0; i<numero_columnas; i++){
            cadena.Anade(letra);
            cin >> letra;
        }
        sopa.Anade(cadena);
        cadena.EliminaTodos();
    }

    cin >> longitud_a_buscar;
    for (int i=0; i<longitud_a_buscar; i++){
        cin >> letra;
        a_buscar.Anade(letra);
    }

    encontrado = sopa.BuscaPalabra(a_buscar);
    cout << encontrado.fila << " " << encontrado.columna;
}
```

¿Qué pasaría al ejecutar el siguiente código?

```
cadena = sopa.Fila(1);  
cadena.Anade('o');           // Se modifica cadena pero NO la sopa
```

Si desde un objeto obj llamamos a un método que devuelve un objeto dev, las modificaciones que hagamos sobre dev no afectan a obj.

En esta asignatura estamos viendo cómo manejar objetos en C++, siempre a través de copias de éstos (al pasarlo como parámetros, al devolverlos en un método, etc)

Otra alternativa es trabajar con referencias a objetos y no con copias. Esta forma de trabajar se consigue en C++ usando **referencias (references)** y **punteros (pointers)** y se verá en el segundo cuatrimestre.

IMPORTANT

El único método que no es trivial es BuscaPalabra y el algoritmo que lo implementaba ya se vio en la página 432

```
struct ParFilaColumna{  
    int fila;  
    int columna;  
};
```

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

```
class SecuenciaCaracteres{  
    ....  
};  
  
class SopaLetras {  
  
private:  
    static const int MAX_FIL = 50;  
    static const int MAX_COL = 40;  
    char matriz_privada[MAX_FIL] [MAX_COL];  
    int util_fil;  
    const int util_col;  
  
public:  
  
    // Prec: 0 < numero_de_columnas <= MAX_COL(40)  
    SopaLetras (int numero_de_columnas)  
        :util_fil(0), util_col(numero_de_columnas)  
    {}  
  
    // Prec: primera_fila.TotalUtilizados() <= MAX_COL(40)  
    SopaLetras (SecuenciaCaracteres primera_fila)  
        :util_fil(0), util_col(primer_fila.TotalUtilizados())  
    {}
```

```
int CapacidadFilas () {
    return MAX_FIL;
}
int FilasUtilizadas () {
    return util_fil;
}
int ColUtilizadas () {
    return util_col;
}
char Elemento (int fila, int columna) {
    return matriz_privada[fila] [columna];
}
SecuenciaCaracteres Fila (int indice_fila) {
    SecuenciaCaracteres fila;

    for (int col = 0; col < util_col; col++)
        fila.Aniaude (matriz_privada[indice_fila] [col]);

    return fila;
}
void Aniaude (SecuenciaCaracteres fila_nueva){
    int numero_columnas_nueva;

    if (util_fil < MAX_FIL){
        numero_columnas_nueva = fila_nueva.TotalUtilizados();

        if (numero_columnas_nueva == util_col){
            for (int col = 0; col < util_col ; col++)
                matriz_privada[util_fil] [col] = fila_nueva.Elemento(col);
            util_fil++;
        }
    }
}
```

```
ParFilaColumna BuscaPalabra (SecuenciaCaracteres a_buscar){  
    bool encontrado, sigo;  
    ParFilaColumna pos_encontrado;  
    int tam_a_buscar;  
  
    encontrado = false;  
    pos_encontrado.fila = pos_encontrado.columna = -1;  
    tam_a_buscar = a_buscar.TotalUtilizados();  
  
    if (tam_a_buscar <= util_col){  
  
        for (int fil = 0; fil < util_fil && !encontrado; fil++){  
  
            for (int col_inicio = 0;  
                 col_inicio + tam_a_buscar <= util_col && !encontrado;  
                 col_inicio++) {  
  
                sigo = true;  
                for (int i=0; i<tam_a_buscar && sigo; i++)  
                    sigo = (matriz_privada[fil] [col_inicio + i]  
                           == a_buscar.Elemento(i));  
                if (sigo) {  
                    encontrado = true;  
                    pos_encontrado.fila = fil;  
                    pos_encontrado.columna = col_inicio;  
                }  
            } // for col_inicio  
        } // for fil  
    } // if (tam_a_buscar <= util_col)  
  
    return pos_encontrado;  
}  
};
```

VI.3.1.2. Tabla dentada usando una matriz

X	X	X	X	X	?	...	?
X	X	?	?	?	?	...	?
X	X	X	X	?	?	...	?
?	?	?	?	?	?	...	?
...
?	?	?	?	?	?	...	?

Una **tabla dentada** es una tabla en la que *las filas pueden tener diferente número de casillas ocupadas*. Su definición podría ser:

```

static const int NUM_FILS = 50; // Filas disponibles
static const int NUM_COLS = 40; // Columnas disponibles

char matriz_privada[NUM_FILS] [NUM_COLS];

// PRE: 0 <= filas_utilizadas <= NUM_FILS

int filas_utilizadas;

// PRE: 0 <= num_cols_utilizadas[i] <= NUM_COLS
//       para i=0,1,...,filas_utilizadas-1
// Número de columnas ocupadas en cada fila

int num_cols_utilizadas[NUM_FILS] = {0};

```

Cada fila puede tener un número de columnas distinto. Lo controlamos con el vector num_cols_utilizadas

1. Como cada fila puede tener un número de casillas ocupadas diferente a las demás, el número de columnas no tiene porqué fijarse con el constructor y podemos usar un constructor sin argumentos para crear una matriz vacía:

```
// Constructor sin argumentos.
```

```
TablaDentadaCaracteres (void)
    : filas_utilizadas(0)
{ }
```

También podemos ofrecer un constructor que recibe la primera fila. El número de casillas de esta fila, no obstante, no supone ninguna restricción para las siguientes. Únicamente tendremos en cuenta que habrá que imponer como precondición que el número de casillas de la primera fila esté en el rango correcto.

```
// Recibe "primera_fila" (una secuencia de caracteres).
// PRE: primera_fila.TotalUtilizados() <= NUM_COLS
```

```
TablaDentadaCaracteres (SecuenciaCaracteres primera_fila)
    : filas_utilizadas(0)
{
    Aniade(primer_fila); // Actualiza "filas_utilizadas"
}
```

El método `Aniade()` se encargará de actualizar correctamente el campo `filas_utilizadas` (en este caso a 1) y el número de casillas de la fila 0 (`num_cols_utilizadas[0]`)

2. Al igual que con las matrices rectangulares, sólo se permitirá añadir filas enteras.

```
// Añade una fila completa (un objeto "SecuenciaCaracteres").
// PRE: fila.TotalUtilizados() = NUM_COLS
// PRE: filas_utilizadas < NUM_FILS

void Aniade (SecuenciaCaracteres fila)
{
    int numero_cols_fila = fila.TotalUtilizados();

    if ((filas_utilizadas < NUM_FILS) &&
        (numero_cols_fila <= NUM_COLS)) {

        for (int c=0; c<numero_cols_fila; c++)
            matriz_privada[filas_utilizadas] [c] = fila.Elemento(c);

        num_cols_utilizadas[filas_utilizadas]=numero_cols_fila;
        filas_utilizadas++;
    }
}
```

3. También podremos recuperar el carácter de una casilla concreta

```
// Devuelve el carácter de la casilla (fila, columna)
// PRE: 0 <= fila < filas_utilizadas
// PRE: 0 <= columna < num_cols_utilizadas[fila]

char Elemento (int fila, int columna)
{
    return (matriz_privada[fila] [columna]);
}
```

o bien una fila entera:

```
// Devuelve una fila completa (un objeto "SecuenciaCaracteres")
// PRE: 0 <= indice_fila < filas_utilizadas

SecuenciaCaracteres Fila (int indice_fila)
{
    SecuenciaCaracteres fila;

    if ((0 <= indice_fila) && (indice_fila < filas_utilizadas)) {

        int num_columnas = num_cols_utilizadas[indice_fila];

        for (int col=0; col<num_columnas; col++)
            fila.Anade(matriz_privada[indice_fila][col]);
    }
    return (fila);
}
```

4. Ofreceremos también los métodos de consulta -triviales- habituales: CapacidadFilas, CapacidadColumnas y FilasUtilizadas **mientras que el número de casillas ocupadas en una fila se consulta en el vector** num_cols_utilizadas

```
// Devuelve el número de casilla ocupadas en una fila
// PRE: 0 <= indice_fila < filas_utilizadas
int ColumnasUtilizadas (int indice_fila)
{
    return (num_cols_utilizadas[indice_fila]);
}
```

A modo de resumen, la clase TablaDentadaCaracteres podría especificarse de acuerdo a esta descripción:

TablaDentadaCaracteres	
- const int	<u>NUM_FILS</u>
- const int	<u>NUM_COLS</u>
- char	matriz_privada[NUM_FILS] [NUM_COLS]
- int	filas_utilizadas
- int	num_cols_utilizadas[NUM_FILS]
+	TablaDentadaCaracteres()
+	TablaDentadaCaracteres(SecuenciaCaracteres primera_fila)
+ int	CapacidadFilas()
+ int	CapacidadColumnas()
+ int	FilasUtilizadas()
+ int	ColumnasUtilizadas(int fila)
+ char	Elemento(int fila, int columna)
+ SecuenciaCaracteres	Fila(int indice_fila)
+ void	Aniade(SecuenciaCaracteres fila_nueva)
+ void	Inserta (int num_fila, SecuenciaCaracteres fila_nueva)
+ void	Elimina (int num_fila)
+ void	EliminaTodos (void)
+ bool	EstaVacia (void)
+ bool	EsIgual (TablaDentadaCaracteres otra_tabla)

Ejemplo. Podemos considerar un objeto de la clase Texto como un conjunto de líneas. Internamente, cada línea corresponde a una fila de una matriz de caracteres.

matriz_privada	util_col
l í n e a s ? ? ? ... ?	6
d e ? ? ? ? ? ? ? ... ?	2
d i s t i n t o ? ... ?	8
t a m a ñ o ? ? ? ... ?	6
? ? ? ? ? ? ... ? ... ?	0
? ? ? ? ? ? ... ? ... ?	0
...	...
? ? ? ? ? ? ... ? ... ?	0

Texto	
- const int	MAX_FIL
- const int	MAX_COL
- char	matriz_privada[MAX_FIL] [MAX_COL]
- int	util_col[MAX_FIL]
- int	util_fil
- void	ReemplazaFila (int fila_a_borrar, int fila_origen)
+ Texto()	
+ int	CapacidadFilas()
+ int	FilasUtilizadas()
+ char	Elemento(int fila, int columna)
+ SecuenciaCaracteres	Fila(int indice_fila)
+ void	Aniade(SecuenciaCaracteres fila_nueva)
+ void	void Inserta(SecuenciaCaracteres fila_nueva, int pos_fila_insercion)

```
int main(){
    const char TERMINADOR_LINEA = '.' ;
    const char TERMINADOR_TEXTO = '#' ;
    char letra;
    SecuenciaCaracteres cadena, a_insertar;
    Texto texto;

    // Formato de entrada:
    // Primera fila.Segunda.Ultima fila.#

    cin >> letra;
    while (letra != TERMINADOR_TEXTO){
        while (letra != TERMINADOR_LINEA){
            cadena.Anade(letra);
            cin >> letra;
        }

        texto.Anade(cadena);
        cadena.EliminaTodos();
        cin >> letra;
    }

    cin >> letra;
    while (letra != TERMINADOR_LINEA){
        a_insertar.Anade(letra);
        cin >> letra;
    }

    texto.Anade(a_insertar);

    cadena = texto.Fila(texto.FilasUtilizadas()-1);
    cout << cadena.ToString();
}
```

Todos los métodos son inmediatos de implementar. Únicamente hay que ir controlando el número de columnas de cada fila.

El único método nuevo es el que inserta una fila entera. El algoritmo sería el siguiente:

Algoritmo: Insertar una fila en una matriz

Recorrer todas las filas desde el final hasta la posición de inserción.
Reemplazar cada fila por la anterior
Volcar la nueva fila en la posición de inserción.

```
class Texto {  
  
private:  
  
    static const int MAX_FIL = 50;  
    static const int MAX_COL = 40;  
    char matriz_privada[MAX_FIL] [MAX_COL];  
    int util_fil;  
    int util_col[MAX_FIL];  
  
    void ReemplazaFila (int fila_a_borrar, int fila_origen){  
  
        int columnas_utilizadas = util_col[fila_origen];  
  
        for (int col = 0; col < columnas_utilizadas; col++)  
            matriz_privada[fila_a_borrar] [col] =  
                matriz_privada[fila_origen] [col];  
  
        util_col[fila_a_borrar] = columnas_utilizadas;  
    }  
}
```

```
public:

// PRE: 0 < numero_de_columnas <= MAX_COL(40)
Texto () : util_fil(0)
{
    for (int i=0; i<MAX_COL; i++)
        util_col[i] = 0;
}

int CapacidadFilas () {
    return MAX_FIL;
}

int FilasUtilizadas () {
    return util_fil;
}

int ColUtilizadas (int indice_fila) {
    return util_col[indice_fila];
}

char Elemento(int fila, int columna){
    return matriz_privada[fila][columna];
}

SecuenciaCaracteres Fila (int indice_fila) {

    SecuenciaCaracteres fila;
    int num_columnas = util_col[indice_fila];

    for (int j = 0; j < num_columnas; j++)
        fila.Anade(matriz_privada[indice_fila][j]);

    return fila;
}
```

```
void Aniade (SecuenciaCaracteres fila_nueva) {  
    if (util_fil < MAX_FIL){  
        int num_cols_nueva = fila_nueva.TotalUtilizados();  
  
        if (num_cols_nueva < MAX_COL){  
            for (int j = 0; j < num_cols_nueva ; j++)  
                matriz_privada[util_fil] [j] = fila_nueva.Elemento(j);  
  
            util_col[util_fil] = num_cols_nueva;  
            util_fil++;  
        }  
    }  
  
    void Inserta (SecuenciaCaracteres fila_nueva, int pos_fila_ins) {  
        int num_cols_nueva = fila_nueva.TotalUtilizados();  
  
        if (num_cols_nueva <= MAX_COL && util_fil < MAX_FIL &&  
            pos_fila_ins >= 0 && pos_fila_ins <= util_fil) {  
  
            for (int i = util_fil ; i > pos_fila_ins ; i--)  
                ReemplazaFila(i, i-1);  
  
            for (int j = 0; j < num_cols_nueva; j++)  
                matriz_privada[pos_fila_ins] [j] = fila_nueva.Elemento(j);  
  
            util_fil++;  
            util_col[pos_fila_ins] = num_cols_nueva;  
        }  
    }  
};
```

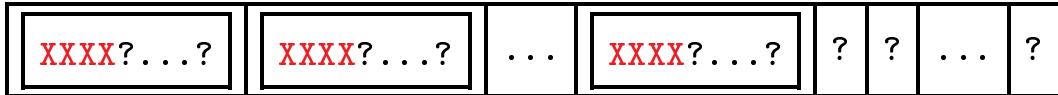
http://decsai.ugr.es/~carlos/FP/V_texto.cpp

VI.3.2. Representación de una tabla con vectores de objetos

En realidad la representación es una **secuencia de secuencias**, esto es, la estructura de datos que se empleará para almacenar los datos será un **vector clásico de objetos de tipo secuencia**.

Si todas las secuencias (las filas, en el contexto de tablas) tienen el mismo número de elementos hablaremos de **tablas rectangulares**, mientras que si cada secuencia puede tener un número arbitrario de elementos hablaremos de **tablas dentadas**.

VI.3.2.1. Tabla rectangular usando un vector de objetos



En una tabla rectangular de caracteres cada **fila** es una secuencia, y todas tienen el mismo número de elementos (**columnas**):

```
static const int NUM_FILS = 50; // "filas" disponibles
static const int NUM_COLS = 40;
```

```
SecuenciaCaracteres vector_privado[NUM_FILS];
```

```
// PRE: 0 <= filas_utilizadas <= NUM_FILS
// PRE: 0 <= cols_utilizadas <= NUM_COLS
int filas_utilizadas;
int cols_utilizadas;
```

Asegurarnos de que todas las secuencias tengan el mismo número de elementos implica un **control estricto en las inserciones/adiciones de filas e impedir borrados de elementos individuales**.

En definitiva, un objeto TablaRectangularTIPO contiene en la parte privada un vector de secuencias, es decir, un vector en el que cada componente es un objeto de la clase SecuenciaTIPO.

El máximo número de columnas viene determinado ahora por la capacidad de la clase SecuenciaTIPO.

La interface de la clase debería ser idéntica a la que usa una matriz clásica para almacenar los elementos. Por ejemplo, la clase TablaRectangularCaracteres se describe:

TablaRectangularCaracteres	
- const int	<u>NUM_FILS</u>
- const int	<u>NUM_COLS</u>
- SecuenciaCaracteres	vector_privado [NUM_FILS]
- int	filas_utilizadas
- int	cols_utilizadas
+	TablaRectangularCaracteres()
+	TablaRectangularCaracteres(int num_columnas)
+	TablaRectangularCaracteres(SecuenciaCaracteres primera_fila)
+ int	CapacidadFilas()
+ int	CapacidadColumnas()
+ int	FilasUtilizadas()
+ int	ColumnasUtilizadas()
+ char	Elemento(int fila, int columna)
+ SecuenciaCaracteres	Fila(int indice_fila)
+ SecuenciaCaracteres	Columna(int indice_fila)
+ void	Aniade(SecuenciaCaracteres fila_nueva)
+ void	Inserta (int num_fila, SecuenciaCaracteres fila_nueva)
+ void	Elimina (int num_fila)
+ void	EliminaTodos (void)
+ bool	EstaVacia (void)
+ bool	EsIgual (TablaRectangularCaracteres otra)

Aunque la implementación de los constructores y los métodos de consulta CapacidadFilas, CapacidadColumnas, FilasUtilizadas y ColumnasUtilizadas pueden ser idénticos, la implementación de los otros métodos será, generalmente, muy diferente.

Algunos ejemplos:

1. Método para recuperar el carácter de una casilla concreta (dentro de la zona ocupada):

```
// Devuelve el carácter de la casilla (fila, columna)
// PRE: 0 <= fila < filas_utilizadas
// PRE: 0 <= columna < cols_utilizadas

char Elemento (int fila, int columna)
{
    return ((vector_privado[fila]).Elemento(columna));
}
```

2. Método para recuperar una fila entera:

```
// Devuelve una fila completa (un objeto "SecuenciaCaracteres")
// PRE: 0 <= indice_fila < filas_utilizadas

SecuenciaCaracteres Fila (int indice_fila)
{
    return (vector_privado[indice_fila]);
}
```

3. Método para añadir una fila:

```
// Añade una fila completa (un objeto "SecuenciaCaracteres").
// PRE: fila.TotalUtilizados() == cols_utilizadas
// PRE: filas_utilizadas < NUM_FILS

void Aniade (SecuenciaCaracteres fila)
{
    int numero_cols_fila = fila.TotalUtilizados();

    if ((filas_utilizadas < NUM_FILS) &&
        (numero_cols_fila == NUM_COLS)) {
        vector_privado[filas_utilizadas] = fila_nueva;
        filas_utilizadas++;
    }
}
```

4. Método para eliminar una fila:

```
// Elimina una fila completa (un objeto "SecuenciaCaracteres").
// PRE: 0 <= num_fila < TotalUtilizados()

void Elimina (int num_fila)
{
    if ((0<=num_fila) && (num_fila <=filas_utilizadas)) {
        // "Desplazar" las filas hacia posiciones bajas.
        // En la posición "num_fila" se copia la que está
        // en la siguiente ("num_fila"+1), ... y en la posición
        // "total_utilizados"-2 se copia la de "total_utilizados"-1.

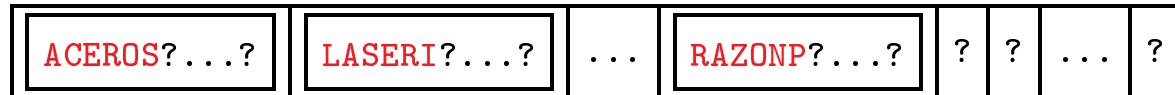
        for (int fil=num_fila; fil<filas_utilizadas-1; fil++)
            vector_privado[fil] = vector_privado[fil+1];
        filas_utilizadas--; // Actualizar el tamaño de la tabla.
    }
}
```

Ejemplo. Implementamos la sopa de letras usando internamente un vector de SecuenciaCaracteres.

```

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   ?   ...   ?
?   ?   ?   ?   ?   ?   ?   ...   ?
...
?   ?   ?   ?   ?   ?   ?   ...   ?

```



La interfaz pública no varía.

SopaLetras	
- const int	MAX_FIL
- const int	util_col
- SecuenciaCaracteres	vector_privado[MAX_FIL]
- int	util_fil
+	SopaLetras (int numero_de_columnas)
+	SopaLetras (SecuenciaCaracteres primera_fila)
+ int	CapacidadFilas ()
+ int	FilasUtilizadas ()
+ int	ColUtilizadas ()
+ char	Elemento (int fila, int columna)
+ SecuenciaCaracteres	Fila (int indice_fila)
+ void	Aniade (SecuenciaCaracteres fila_nueva)
+ ParFilaColumna	BuscaPalabra (SecuenciaCaracteres a_buscar)

Como la interfaz pública no ha cambiado, la función `main` es la misma (página 667). Al igual que pasaba en la primera versión:

```
cadena = sopa.Fila(1);
cadena.Anade('o');           // Se modifica cadena pero NO la sopa
```

Si un método devuelve una copia de una componente de un vector dato miembro, las modificaciones que se realicen posteriormente sobre la copia no afectan a la componente original.

Implementación:

Vemos primero los métodos más básicos.

```
class SopaLetras {

private:

    static const int MAX_FIL = 50;
    SecuenciaCaracteres vector_privado[MAX_FIL];
    int util_fil;
    const int util_col;

public:

    // PRE: 0 < numero_de_columnas <= Capacidad de SecuenciaCaracteres
    SopaLetras (int numero_de_columnas)
        : util_fil(0), util_col(numero_de_columnas) { }

    // No hay ninguna precondición
    SopaLetras (SecuenciaCaracteres primera_fila)
        : util_fil(0), util_col(primera_fila.TotalUtilizados()) { }
```

```
int CapacidadFilas () {
    return MAX_FIL;
}
int FilasUtilizadas () {
    return util_fil;
}
int ColUtilizadas () {
    return util_col;
}
char Elemento (int fila, int columna) {
    return vector_privado[fila].Elemento(columna);
}
SecuenciaCaracteres Fila (int indice_fila){
    return vector_privado[indice_fila];
}
void Aniade (SecuenciaCaracteres fila_nueva) {

    if (util_fil < MAX_FIL){
        int num_cols_fila_nueva = fila_nueva.TotalUtilizados();

        if (num_cols_fila_nueva == util_col){
            vector_privado[util_fil] = fila_nueva;
            util_fil++;
        }
    }
}

ParFilaColumna BuscaPalabra (SecuenciaCaracteres a_buscar) {
    .....
}
};
```

¿Cómo implementamos el método BuscaPalabra?

- ▷ De forma similar a como se hizo en la versión que usaba una matriz de doble corchete.
- ▷ Mejor aún. ¿No es lógico que la comparación entre dos secuencias de caracteres sea algo que queramos hacer en otros contextos distintos de la sopa de letras? Por tanto, añadimos un método a la clase SecuenciaCaracteres:

SecuenciaCaracteres
.....
+ int PosContiene(SecuenciaCaracteres pequenia)

Fomentad la creación de métodos genéricos situándolos en aquellas clases que previsiblemente vayan a reutilizarse en más ocasiones.

Hemos cambiado la interfaz pública de SecuenciaCaracteres pero no la de la clase Sopa por lo que la función main es la misma (página 667).

```
class SopaLetras {  
    ....  
    ParFilaColumna BuscaPalabra (SecuenciaCaracteres a_buscar) {  
  
        ParFilaColumna pos_encontrado = {-1, -1};  
  
        int tamanio_a_buscar = a_buscar.TotalUtilizados();  
  
        if (tamanio_a_buscar <= util_col) {  
  
            bool encontrado = false;  
  
            for (int i=0; i < util_fil && !encontrado; i++){  
  
                SecuenciaCaracteres fila = vector_privado[i];  
                int col_encontrado = fila.PosContiene(a_buscar);  
  
                if (col_encontrado != -1) {  
                    encontrado = true;  
                    pos_encontrado.fila = i;  
                    pos_encontrado.columna = col_encontrado;  
                }  
            }  
        }  
  
        return pos_encontrado;  
    }  
    ....  
};
```

Algoritmo: Buscar una subcadena peque dentro de grande

Recorrer las componentes i de grande
 hasta que se terminen o hasta que se encuentre peque
 Si grande[i] == peque[0]
 Comprobar el resto de componentes de peque.
 Si todas se encuentran,
 salir recordando la posición inicial i

```
class SecuenciaCaracteres {
    .....
    // Comprueba si pequenia está dentro de la cadena actual
    // Si pequenia está vacía devuelve -1

    int PosContiene (SecuenciaCaracteres pequenia) {

        int pos_contiene = -1;

        int utilizados_pequenia = pequenia.TotalUtilizados();
        int tope = total_utilizados - utilizados_pequenia;
        bool encontrado = false;

        int inicio = 0;
        while (inicio < tope && !encontrado) {

            bool van_siendo_iguales = true;
            int i = inicio;
            while (i < utilizados_pequenia && van_siendo_iguales) {
                if (vector_privado[inicio + i] != pequenia.Elemento(i))
                    van_siendo_iguales = false;
                else i++;
            }
        }
    }
}
```

```
    if (van_siendo_iguales) {
        encontrado = true;
        pos_contiene = inicio;
    }
    else inicio++;

}

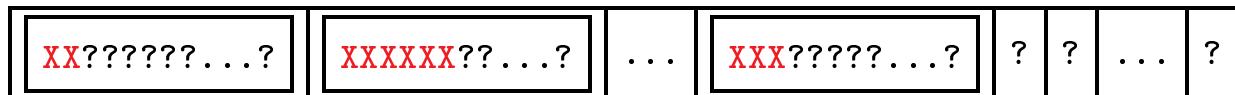
return pos_contiene;
}

.....
};
```

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

VI.3.2.2. Tabla dentada usando un vector de objetos

El tipo de estructura que vamos a construir es similar a la tabla rectangular con un vector de objetos, sólo que ahora no nos preocupamos de que todas las filas tengan el mismo número de columnas.



Si se trata de una tabla dentada de caracteres, las “filas” (cada secuencia guardada `vector_privado`) podrán tener diferente número de casillas ocupadas, por lo que no tiene sentido emplear `cols_utilizadas`.

El control acerca del número de elementos que tendrá cada “fila” se realizará en la clase `SecuenciaCaracteres` ya que es ahí dónde está definida **TAMANIO** (la constante que hace el papel de `NUM_COLS`).

```
static const int NUM_FILS = 50; // "filas" disponibles

SecuenciaCaracteres vector_privado[NUM_FILS];

// PRE: 0 <= filas_utilizadas <= NUM_FILS

int filas_utilizadas; // Número de "filas" ocupadas
```

La interface de la clase debería ser idéntica a la que usa una matriz clásica para almacenar los elementos. Por ejemplo, la clase TablaDentadaCaracteres se describe:

TablaDentadaCaracteres	
- const int	NUM_FILS
- SecuenciaCaracteres	vector_privado[NUM_FILS]
- int	filas_utilizadas
+	TablaDentadaCaracteres()
+	TablaDentadaCaracteres(SecuenciaCaracteres primera_fila)
+ int	CapacidadFilas()
+ int	CapacidadColumnas()
+ int	FilasUtilizadas()
+ int	ColumnasUtilizadas(int indice_fila)
+ char	Elemento(int fila, int columna)
+ SecuenciaCaracteres	Fila(int indice_fila)
+ void	Aniade(SecuenciaCaracteres fila_nueva)
+ void	Inserta (int num_fila, SecuenciaCaracteres fila_nueva)
+ void	Elimina (int num_fila)
+ void	EliminaTodos (void)
+ bool	EstaVacia (void)
+ bool	EsIgual (TablaDentadaCaracteres otra)

Implementación:

1. Constructor sin argumentos (tabla vacía)

```
TablaDentadaCaracteres (void)
{
    filas_utilizadas = 0;
}
```

2. Constructor a partir de una secuencia (primera fila)

```
TablaDentadaCaracteres (SecuenciaCaracteres la_secuencia)
{
    vector_privado[0] = la_secuencia;
    total_utilizados = 1;
}
```

3. Los métodos de consulta CapacidadFilas y FilasUtilizadas son triviales. No ocurre lo mismo con CapacidadColumnas y ColumnasUtilizadas:

```
// Devuelve el máximo número de casillas de una fila
// Observe la llamada a Capacidad sobre un objeto anónimo vacío
int CapacidadColumnas ()
{
    return (SecuenciaCaracteres().Capacidad());
}

// Devuelve el número de casilla ocupadas en una fila
// PRE: 0 <= indice_fila < filas_utilizadas
int ColumnasUtilizadas (int indice_fila)
{
    return (vector_privado[indice_fila].TotalUtilizados());
}
```

4. Añadir una fila a la tabla:

```
// Añade una fila completa (un objeto "SecuenciaCaracteres").  
// PRE: filas_utilizadas < NUM_FILS  
void Aniade (SecuenciaCaracteres fila)  
{  
    if (total_utilizados < TAMANIO) {  
        vector_privado[total_utilizados] = fila;  
        total_utilizados++;  
    }  
}
```

5. Eliminar una fila de la tabla:

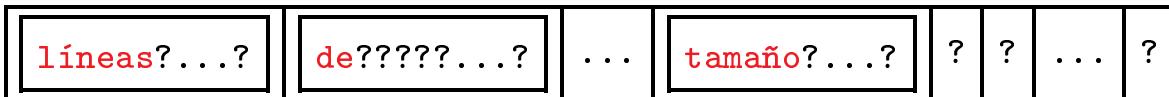
```
// Elimina una fila completa (un objeto "SecuenciaCaracteres").  
// PRE: 0 <= indice_fila < total_utilizados  
void Elimina (int indice_fila)  
{  
    if ((indice_fila >= 0) && (indice_fila < total_utilizados)) {  
  
        int tope = total_utilizados-1; // posición del último  
  
        for (int i = indice_fila ; i < tope ; i++)  
            vector_privado[i] = vector_privado[i+1];  
  
        total_utilizados--;  
    }  
}
```

Ejemplo. Implementamos la clase Texto usando internamente un vector de SecuenciaCaracteres.

Cada fila será un objeto de la clase SecuenciaCaracteres por lo que la gestión de la longitud de cada una de ellas la haremos en la clase SecuenciaCaracteres y no en la clase Texto. Por tanto, ya no es necesario el dato miembro vector util_col de Texto (ver página 679)

```
fila = tabla.Fila(0);
columnas_usadas = fila.TotalUtilizados();
```

l	í	n	e	a	s	?	?	?	...	?
d	e	?	?	?	?	?	?	?	...	?
d	i	s	t	i	n	t	o	?	...	?
t	a	m	a	ñ	o	?	?	?	...	?
?	?	?	?	?	?	?	...	?	...	?
?	?	?	?	?	?	?	...	?	...	?
...										
?	?	?	?	?	?	?	...	?	...	?



La interfaz pública no varía.

Texto	
- const int	MAX_FIL
- SecuenciaCaracteres	vector_privado[MAX_FIL]
- int	util_fil
+ Texto()	
+ int	CapacidadFilas()
+ int	FilasUtilizadas()
+ char	Elemento(int fila, int columna)
+ SecuenciaCaracteres	Fila(int indice_fila)
+ void	Aniade(SecuenciaCaracteres fila_nueva)
+ void	void Inserta(SecuenciaCaracteres fila_nueva, int pos_fila_insercion)

Como la interfaz pública no ha cambiado, la función main es la misma (página 679). Al igual que pasaba en la primera versión:

```
cadena = texto.Fila(1);
cadena.Aniade('o');           // Se modifica cadena pero NO el texto
```

Implementación:

```
class Texto {

private:
    static const int MAX_FIL = 50;
    SecuenciaCaracteres vector_privado[MAX_FIL];
    int util_fil;
```

```
public:
```

```
// PRE: 0 < numero_de_columnas <= Capacidad de SecuenciaCaracteres
Texto() : util_fil (0) { }

int CapacidadFilas () {
    return MAX_FIL;
}

int FilasUtilizadas () {
    return util_fil;
}

int ColUtilizadas (int indice_fila) {
    return vector_privado[indice_fila].TotalUtilizados();
}

char Elemento (int fila, int columna) {
    return vector_privado[fila].Elemento(columna);
}

SecuenciaCaracteres Fila (int indice_fila) {
    return vector_privado[indice_fila];
}

void Aniade (SecuenciaCaracteres fila_nueva) {
    if (util_fil < MAX_FIL){
        vector_privado[util_fil] = fila_nueva;
        util_fil++;
    }
}
```

```
void Inserta (SecuenciaCaracteres fila_nueva,
              int pos_fila_insercion) {

    if (util_fil < MAX_FIL && pos_fila_insercion >= 0 &&
        pos_fila_insercion <= util_fil){

        for (int i = util_fil ; i > pos_fila_insercion ; i--)
            vector_privado[i] = vector_privado[i-1];

        vector_privado[pos_fila_insercion] = fila_nueva;
        util_fil++;
    }

};

};
```

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

VI.4. Diseño de una solución (Ampliación)

VI.4.1. Funciones globales versus métodos

Hemos visto que las funciones pueden definirse:

- ▷ Encapsuladas dentro de una clase (métodos)
- ▷ Globales, fuera de las clases

Como norma general, *no usaremos funciones globales*. Sin embargo, en ocasiones, puede justificarse el uso de éstas. Las usaremos si son *funciones auxiliares genéricas*

- ▷ que podamos utilizar en muchas clases diferentes, o
- ▷ trabajen sobre tipos básicos predefinidos.

► **Usando un método privado** 

```
class Punto2D {  
  
private:  
    double abscisa;  
    double ordenada;  
  
    bool SonIguales (double uno, double otro) {  
        return fabs(uno-otro) <= UMBRAL  
    }  
  
public:  
    Punto2D (double abscisaPunto, double ordenadaPunto)  
        : abscisa(abscisaPunto), ordenada(ordenadaPunto) {}  
  
    double GetAbscisa () {  
        return abscisa;  
    }  
    double GetOrdenada () {  
        return ordenada;  
    }  
    bool EsIgual (Punto2D otro_punto) {  
        return SonIguales(abscisa, otro_punto.GetAbscisa()) &&  
               SonIguales(ordenada, otro_punto.GetOrdenada());  
    }  
};
```

Esta aproximación es correcta, pero ¿y si necesitamos comparar dos doubles en otras clases? Habría que repetir el código del método SonIguales en cada una de estas clases.

► **Usando una función global** 

```
bool SonIguales (double uno, double otro)  {
    return fabs(uno-otro) <= UMBRAL;
}

class Punto2D {

private:
    double abscisa;
    double ordenada;

public:
    Punto2D(double abscisaPunto, double ordenadaPunto)
        : abscisa(abscisaPunto), ordenada (ordenadaPunto) { }

    double GetAbscisa () {
        return abscisa;
    }

    double GetOrdenada () {
        return ordenada;
    }

    bool EsIgual (Punto2D otro_punto){
        return    SonIguales(abscisa, otro_punto.GetAbscisa()) &&
                SonIguales(ordenada, otro_punto.GetOrdenada());
    }
};

int main(){
    <Aquí también puedo usar directamente SonIguales>
```

Puede aprovechar la función SonIguales en otras aplicaciones.

```
bool SonIguales(double uno, double otro) {  
    return fabs(uno-otro) <= UMBRAL  
}  
  
class TransaccionBancaria {  
public:  
    double Importe(){  
        .....  
    }  
    .....  
};  
int main() {  
    TransaccionBancaria una_transaccion, otra_transaccion;  
    bool son_iguales;  
    .....  
    son_iguales = SonIguales(una_transaccion.Importe(),  
                            otra_transaccion.Importe());  
    .....
```

Otros ejemplos de funciones globales:

```
int MaximoComunDivisor (int un_entero, int otro_entero);  
double Maximo (double un_real, double otro_real);  
char Transforma_a_Mayuscula (char un_caracter);
```

► **Usando un método público** ☹

¿Nos vale poner el método `SonIguales` como **público** y así poder usarlo en otros sitios? Poder, se puede, pero no tiene sentido:

```
class Punto2D{  
private:  
    double abscisa;  
    double ordenada;  
    .....  
public:  
    bool SonIguales(double uno, double otro)  {  
        return fabs(uno-otro) <= UMBRAL;  
    }  
    .....  
};  
class TransaccionBancaria{  
private:  
    double importe;  
public:  
    double Importe() {  
        return importe;  
    }  
    .....  
};  
int main(){  
    TransaccionBancaria trans1, trans2;  
    Punto2D punto_absurdo (4.3, 5.8);  
    bool son_iguales;  
    .....  
    son_iguales = punto_absurdo.SonIguales (trans1.Importe(),  
                                             trans2.Importe());
```

VI.4.2. Fábricas de objetos (Ampliación)

Sabemos que no podemos mezclar en una misma clase cómputos con E/S. ¿Cómo encapsularíamos las tareas de leer o imprimir los datos miembro de una clase?

Las tareas necesarias para realizar las operaciones de E/S de los datos de un objeto, se realizarán fuera de la clase.

Para la entrada de datos, dada una clase C , será usual crear otra clase fábrica (factory) que construya objetos de C a través de un método (el método devolverá un objeto de C).

Para mostrar los datos de una clase aconsejamos escribir un método `ToString` para serializar objetos formando un dato string. Entonces, fuera de la clase se trabajará con el string que representa el estado del objeto.

Ejemplo. Definimos clases para *leer* puntos y rectas.

LectorPuntos
- string mensaje
+ LectorPuntos (string el_mensaje)
+ void ImprimeMensajeEntrada ()
+ Punto2D Lee ()

LectorRectas
- string mensaje
+ LectorRectas (string el_mensaje)
+ void ImprimeMensajeEntrada ()
+ Recta Lee ()

y en las clases Punto2D y Recta escribimos el método ToString:

```
class Punto2D {
private:
    double abscisa, ordenada;
public:
    ...
    string ToString () {
        return "("+to_string(abscisa) + "," +to_string(ordenada)+")";
    }
};

class LectorPuntos {
private:
    string mensaje;
public:
    LectorPuntos (string el_mensaje) : mensaje (el_mensaje) { }

    void ImprimeMensajeEntrada (){
        cout << mensaje;
    }
};
```

```
Punto2D Lee () {
    double abscisa, ordenada;
    cin >> abscisa;
    cin >> ordenada;

    Punto2D punto(abscisa, ordenada);
    return punto;
}

};

class Recta {
private:
    double A, B, C;
public:
    ...
    string ToString () {
        return + to_string(A) + " x + " + to_string(B) +
               " y + " + to_string(C) + " = 0";
    }
};

class LectorRectas {
private:
    string mensaje;
public:
    LectorRectas (string el_mensajes) : mensaje (el_mensajes) { }

    void ImprimeMensajeEntrada () {
        cout << mensaje;
    }
}
```

```
Recta Lee (){

    double A, B, C;
    cin >> A;
    cin >> B;
    cin >> C;

    Recta recta(A, B, C);
    return recta;
}

};

int main(){

    LectorPuntos lector_puntos ("Introduzca coordenadas del punto: ");
    LectorRectas lector_rectas ("Introduzca coeficientes de la recta: ");

    lector_puntos.ImprimeMensajeEntrada();
    Punto2D un_punto (lector_de_puntos.Lee()); // Constructor de copia

    cout << "El punto es = " << un_punto.ToString() << endl;

    lector_rectas.ImprimeMensajeEntrada();
    Recta una_recta (lector_de_rectas.Lee()); // Constructor de copia

    cout << "La recta es = " << una_recta.ToString() << endl;

    if (una_recta.Contiene(un_punto))
        cout << "La recta contiene al punto.";
    else
        cout << "La recta no contiene al punto.";
```

VI.5. Tratamiento de errores con excepciones

Esta sección no entra en el examen



Supongamos que se violan las precondiciones de un método ¿Cómo lo notificamos al cliente del correspondiente objeto?

```
class SecuenciaCaracteres{  
private:  
    static const int TAMANIO = 50;  
    char vector_privado[TAMANIO];  
    int total_utilizados;  
public:  
    .....  
    void Aniade(char nuevo){  
        if (total_utilizados < TAMANIO){  
            vector_privado[total_utilizados] = nuevo;  
            total_utilizados++;  
        }  
        else  
            cout << "No hay componentes suficientes";  
    }  
};  
int main(){  
    SecuenciaCaracteres cadena;  
  
    cadena.Aniade('h');  
    .....  
    cadena.Aniade('a'); // Si ya no cabe => "No hay componentes suficientes"  
}
```



Esta solución rompe la norma básica de no mezclar C-E/S

El método Aniade ha sido el encargado de tratar el error producido por la violación de la precondición. Esto es poco **flexible** 😞

Si se detecta un error en un método (como la violación de una precondición) la respuesta (acciones a realizar para tratar el error) debe realizarse fuera de dicho método

La solución clásica es devolver un código de error (bool, char, entero, enumerado, etc)

```
class SecuenciaCaracteres{
private:
    static const int TAMANIO = 50;
    char vector_privado[TAMANIO];
    int total_utilizados;
public:
    .....
    int Aniade(char nuevo){
        int error = 0;

        if (total_utilizados < TAMANIO){
            vector_privado[total_utilizados] = nuevo;
            total_utilizados++;
        }
        else
            error = 1;

        return error;
    }
};
```

```
int main(){
    SecuenciaCaracteres cadena;

    .....
    if (cadena.Aniaude('h') == 1)
        cout << "No hay componentes suficientes";
    else{
        <sigo la ejecución del programa>
    };
}
```

Problema: El excesivo anidamiento oscurece el código:

```
int main(){
    SecuenciaCaracteres cadena;

    if (cadena.Aniaude('h') == 1)
        cout << "No hay componentes suficientes";
    else{
        if (cadena.Aniaude('o') == 1)
            cout << "No hay componentes suficientes";
        else
            if (cadena.Aniaude('y') == 1)
                cout << "No hay componentes suficientes";
            else
                <sigo la ejecución del programa>
    };
}
```

Solución: Utilizar el mecanismo de gestión de excepciones.

En PDO una **excepción (exception)** es un **objeto** que contiene información que es traspasada desde el sitio en el que ocurre un problema a otro sitio en el que se tratará dicho problema.

La idea es la siguiente:

- ▷ Un método **genera una excepción** con información sobre un error.
La generación de la excepción se realiza a través del operador `throw`
- ▷ El flujo de control sale del método y va directamente a un lugar en el que se **gestiona el error**.
Dicho lugar queda determinado por un bloque `try - catch`.

En C++, una excepción puede ser de cualquier tipo de dato: entero, real, nuestras propias clases, etc. Por ahora, utilizaremos clases de excepción estándar, incluidos en la biblioteca `stdexcept`, como por ejemplo:

- ▷ Clase `logic_error`:
Se usa para notificar que ha habido un error *lógico* como por ejemplo la violación de una precondición de un método.
- ▷ Clase `runtime_error`:
Se usa para notificar que ha habido un error *de ejecución* como por ejemplo una división entera entre cero o un acceso a un fichero inexistente.

Hay una sobrecarga de los constructores de dichas clases que acepta una cadena de caracteres. En dicha cadena indicaremos el error que se ha producido.

Una vez creado el objeto de excepción se consulta dicha cadena con el método `what()`.

```
#include <stdexcept>
class SecuenciaCaracteres{
private:
    static const int TAMANIO = 50;
    char vector_privado[TAMANIO];
    int total_utilizados;
public:
    .....
    void Aniade(char nuevo){
        if (total_utilizados < TAMANIO){
            vector_privado[total_utilizados] = nuevo;
            total_utilizados++;
        }
        else{
            logic_error objeto_excepcion("No hay componentes suficientes");
            throw objeto_excepcion;
        }
    }
};

int main(){
    SecuenciaCaracteres cadena;

    try{
        cadena.Aniade('h');
        .....
        cadena.Aniade('a'); // Si ya no cabe =>
                            // El flujo de control salta al catch
        .....
    }
    catch(logic_error excepcion){
        cout << "Error lógico: " << excepcion.what();
    }
}
```

Por ahora, el tratamiento del error dentro del catch únicamente consiste en informar con un cout.

Si no construimos el bloque try catch y salta una excepción, el programa abortará y perderemos el control de su ejecución.

```
int main(){
    SecuenciaCaracteres cadena;

    cadena.Anade('h');
    .....
    cadena.Anade('a'); // Si ya no cabe => El programa aborta
```

Si se desea, las dos sentencias:

```
logic_error objeto_excepcion("No hay componentes suficientes");
throw objeto_excepcion;
```

pueden resumirse en una sola:

```
throw logic_error("No hay componentes suficientes");
```

Quedaría así:

```
#include <stdexcept>
class SecuenciaCaracteres{
private:
    static const int TAMANIO = 50;
    char vector_privado[TAMANIO];
    int total_utilizados;
public:
    .....
    void Aniade(char nuevo){
        if (total_utilizados < TAMANIO){
            vector_privado[total_utilizados] = nuevo;
            total_utilizados++;
        }
        else
            throw logic_error("No hay componentes suficientes");
            // Se crea el objeto de la clase logic_error
            // El flujo de control sale del método
    }
};
```

El resto de métodos también pueden generar excepciones:

```
class SecuenciaCaracteres{  
    .....  
    void Inserta(int pos_insercion, char valor_nuevo){  
        if ((total_utilizados < DIM) && (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++;  
        }  
        else  
            throw logic_error("Posición de inserción inválida");  
    }  
};  
int main(){  
    SecuenciaCaracteres cadena;  
  
    try{  
        cadena.Anade('h');  
        .....  
        cadena.Inserta(60, 'a');  
        .....  
    }  
    catch(logic_error excepcion){  
        cout << "Error lógico: " << excepcion.what();  
    }  
}
```

Según sea la excepción que se lance, el método what() devolverá Posición de inserción inválida o bien No hay componentes suficientes.

En el mismo bloque try-catch podemos capturar excepciones generadas por métodos de distintas clases:

```
class SecuenciaCaracteres{  
    .....  
};  
class SopaLetras{  
    .....  
    void Aniade(SecuenciaCaracteres fila_nueva){  
        if (util_fil < MAX_FIL){  
            vector_privado[util_fil] = fila_nueva;  
            util_fil++;  
        }  
        else  
            throw logic_error("No hay más filas disponibles");  
    }  
};  
int main(){  
    SecuenciaCaracteres cadena;  
    SopaLetras sopa;  
  
    try{  
        cadena.Aniade('h');  
        .....  
        sopa.Aniade(cadena);  
        .....  
    }  
    catch(logic_error excepcion){  
        cout << "Error lógico: " << excepcion.what();  
    }  
}
```

¿Y si se generan excepciones de tipos distintos? Hay que añadir el catch correspondiente.

```
class LectorSecuenciaCaracteres{
private:
    const char terminador;
    const string nombre_fichero;
    ifstream lector; // Para leer de un fichero
public:
    LectorSecuenciaCaracteres(string nombre_fichero_entrada,
                                char terminador_datos)
        :terminador(terminador_datos),
         nombre_fichero(nombre_fichero_entrada)
    {
    }
    void AbreFichero(){
        lector.open(nombre_fichero);

        if (lector.fail()){
            string mensaje_error = "Error en la apertura del fichero " +
                                  nombre_fichero;
            throw runtime_error(mensaje_error);
        }
    }
    void CierraFichero(){....}
    SecuenciaCaracteres LeeSecuencia(){....}
    bool HayDatos(){....}
};

int main(){
    SecuenciaCaracteres cadena;
    SopaLetras sopa;
    const char terminador = '#';
    LectorSecuenciaCaracteres input_cadena("DatosEntrada.txt", terminador);
```

```
try{
    input_vector.AbreFichero();
    .....
    cadena = input_cadena.LeeSecuencia();
    .....
    sopa.Aniaude(cadena);
    .....
}
catch(logic_error excepcion){
    cout << "Error lógico: " << excepcion.what();
}
catch(runtime_error excepcion){
    cout << "Error de ejecución: " << excepcion.what();
}
```

A tener en cuenta:

- ▷ Si se produce una excepción de un tipo, el flujo de control entra en el `catch` que captura el correspondiente tipo. No entra en los otros bloques `catch`.
- ▷ Si se omite el `catch` correspondiente a un tipo y se produce una excepción de dicho tipo, el programa abortará.

El tratamiento de errores de programación usando el mecanismo de las excepciones se basa en asociar el error con un TIPO de excepción

Supongamos que un método A llama a otro B y éste a otro C. ¿Qué pasa si salta una excepción en C?

El flujo de control va saliendo de todas las llamadas de los métodos, hasta llegar a algún sitio en el que haya una sentencia `try-catch`. Este proceso se conoce como **stack unwinding** (*limpieza de la pila* sería una posible traducción). Lo recomendable es lanzar la excepción en los últimos niveles y capturarla en los primeros (`main`) → **Throw early catch late**

Consideraciones finales:

- ▷ El uso de excepciones modifica el flujo de control del programa. Un mal programador podría forzar su uso en sustitución de estructuras de control básicas como las condicionales.
Siempre reservaremos el uso de excepciones para programar casos excepcionales, como por ejemplo, cuando la violación de las precondiciones de un método pueda tener efectos potencialmente peligrosos (recordar lo visto en la página **560**)
- ▷ En C++, una excepción puede ser de cualquier tipo de dato: entero, real, nuestras propias clases, etc.
Al crear nuestras propias clases de excepción, podremos construir objetos con más información que un simple `string` sobre el error producido.
- ▷ Cuando se vea el concepto de *Herencia* se verá cómo pueden capturarse diferentes tipos de excepciones en un único `catch`. Para ello, tendremos varias clases de excepción que heredan de otra clase *base*.

VI.6. Ciclo de vida del software (Ampliación)

Esta sección no entra en el examen



Fases esenciales durante el desarrollo del software:

▷ **Planificación (planning)**

Comprende el **análisis de requisitos (requirements analysis)** para establecer las necesidades del cliente, así como el análisis de costes y riesgos.

▷ **Análisis (analysis) y diseño (design)**

En esta fase se analiza qué metodología de programación (procedural, funcional, orientada a objetos, etc) se adapta mejor a la resolución del problema y se diseña la arquitectura de la solución.

En el tema V se verán algunos conceptos relacionados con el diseño de una solución usando PDO.

También se evalúan los recursos hardware/software disponibles.

▷ **Implementación (Implementation)**

Una vez realizado el diseño, se procede a su implementación o **codificación (coding)**.

▷ **Validación (validation) y Verificación (verification)**

Durante la validación comprobamos que el software construido cumple los requerimientos del cliente (**hemos construido el producto adecuado**). En esta fase es necesario interactuar con el cliente.

Durante la verificación comprobamos que el software construido no tiene errores (**hemos construido adecuadamente el producto**).

En esta fase no se interactúa con el cliente. Se desarrollan una serie o batería de **pruebas (tests)** para comprobar el correcto fun-

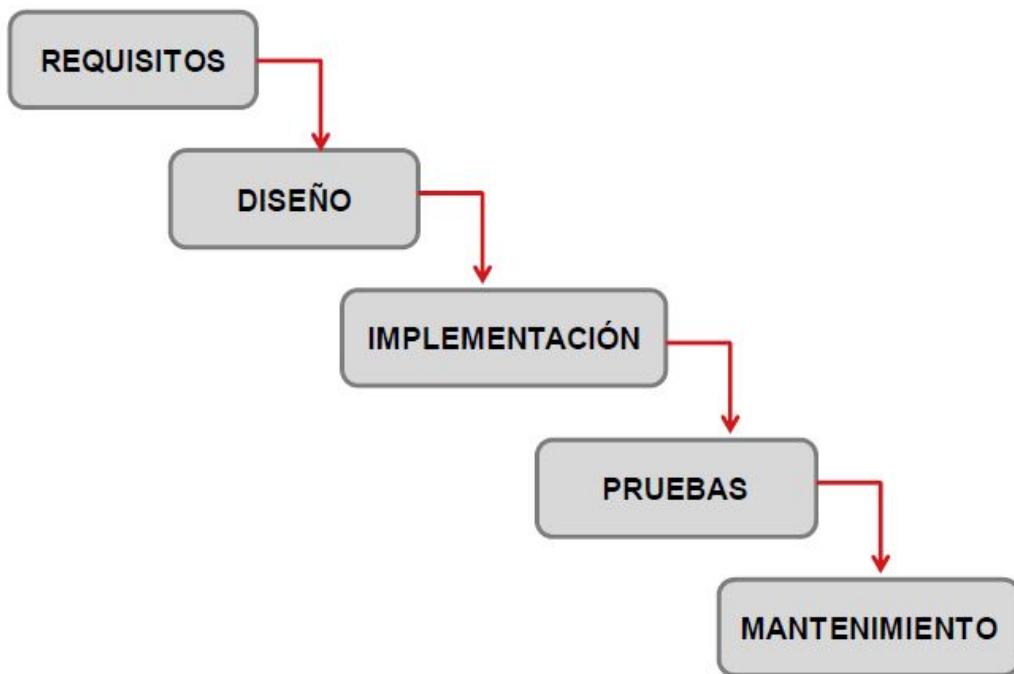
cionamiento:

- **Pruebas de unidad (unit testing)** . Son pruebas dirigidas a comprobar el correcto funcionamiento de un módulo (por ejemplo, una función o un objeto)
- **Pruebas de integración (integration tests)** . Son pruebas dirigidas a comprobar la correcta integración entre los módulos desarrollados.

▷ **Desarrollo (deployment) y mantenimiento (maintenance)** .

Una vez construido el software, se procede a su distribución en un entorno de producción, generación de documentación, programas de aprendizaje, marketing, etc. Durante la explotación real, pueden cambiar o surgir nuevos requerimientos que necesiten volver al inicio del ciclo de vida.

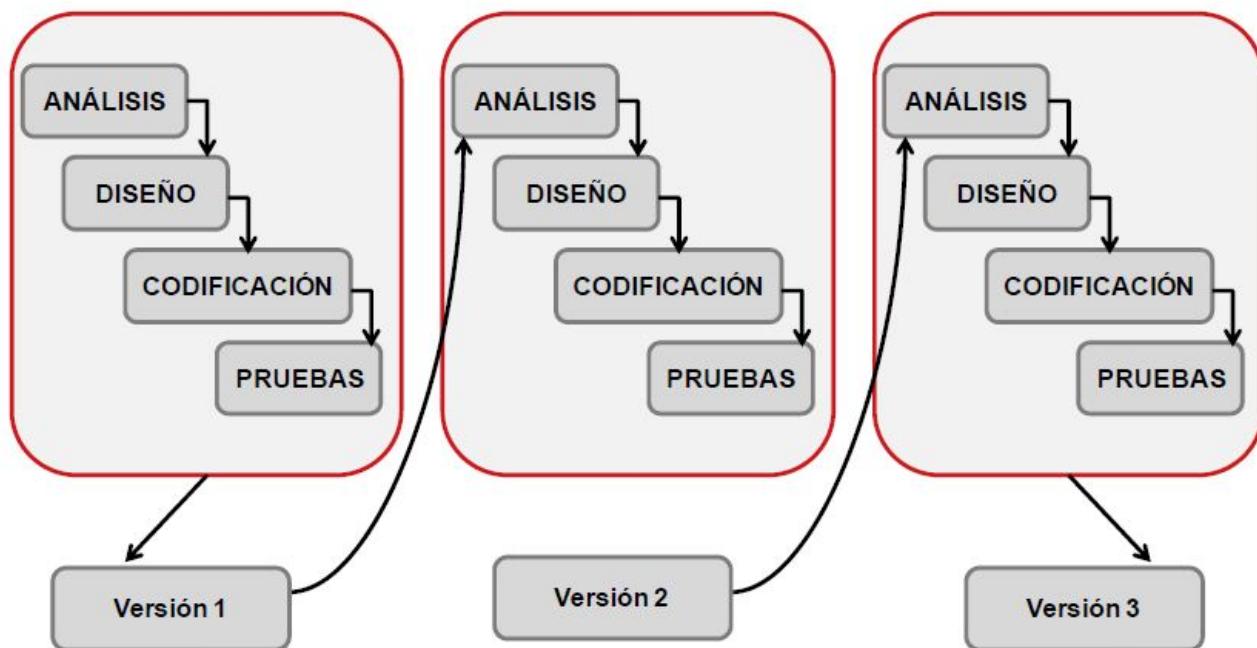
El ciclo de vida básico sería el **modelo en cascada (waterfall model)** :



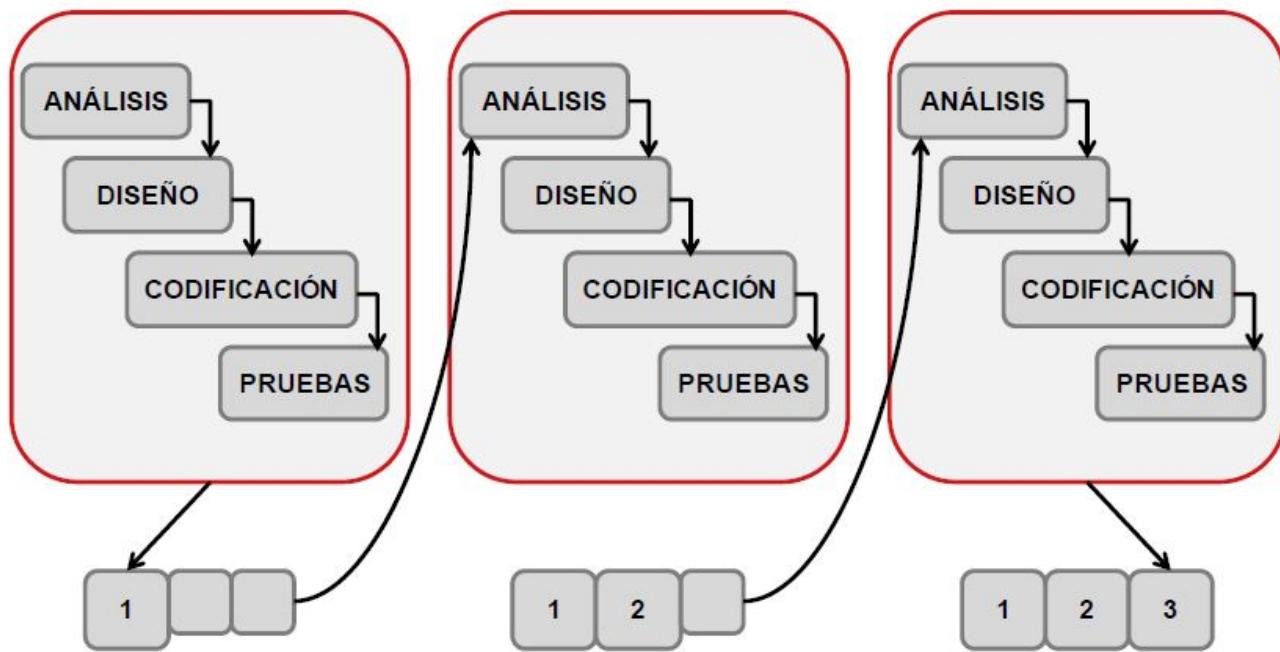
Sin embargo, en el desarrollo de cualquier proyecto software de mediana envergadura deben tenerse en cuenta otros factores como:

- ▷ Análisis de riesgos
- ▷ Tiempo dedicado al desarrollo de cada fase
- ▷ Interacción entre los equipos de desarrollo
- ▷ Interacción con el cliente
- ▷ Secuenciación temporal entre las fases
- ▷ Necesidad de obtener resultados (prototipos) rápidamente

La inclusión de estos factores lleva a considerar ciclos de vida más complejos como por ejemplo los *modelos iterativos (iterative models)* :



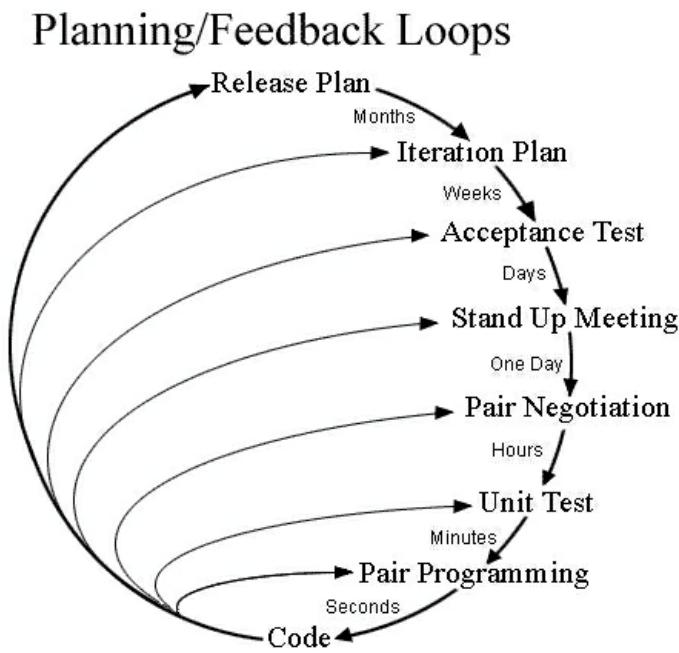
modelos incrementales (incremental models) :



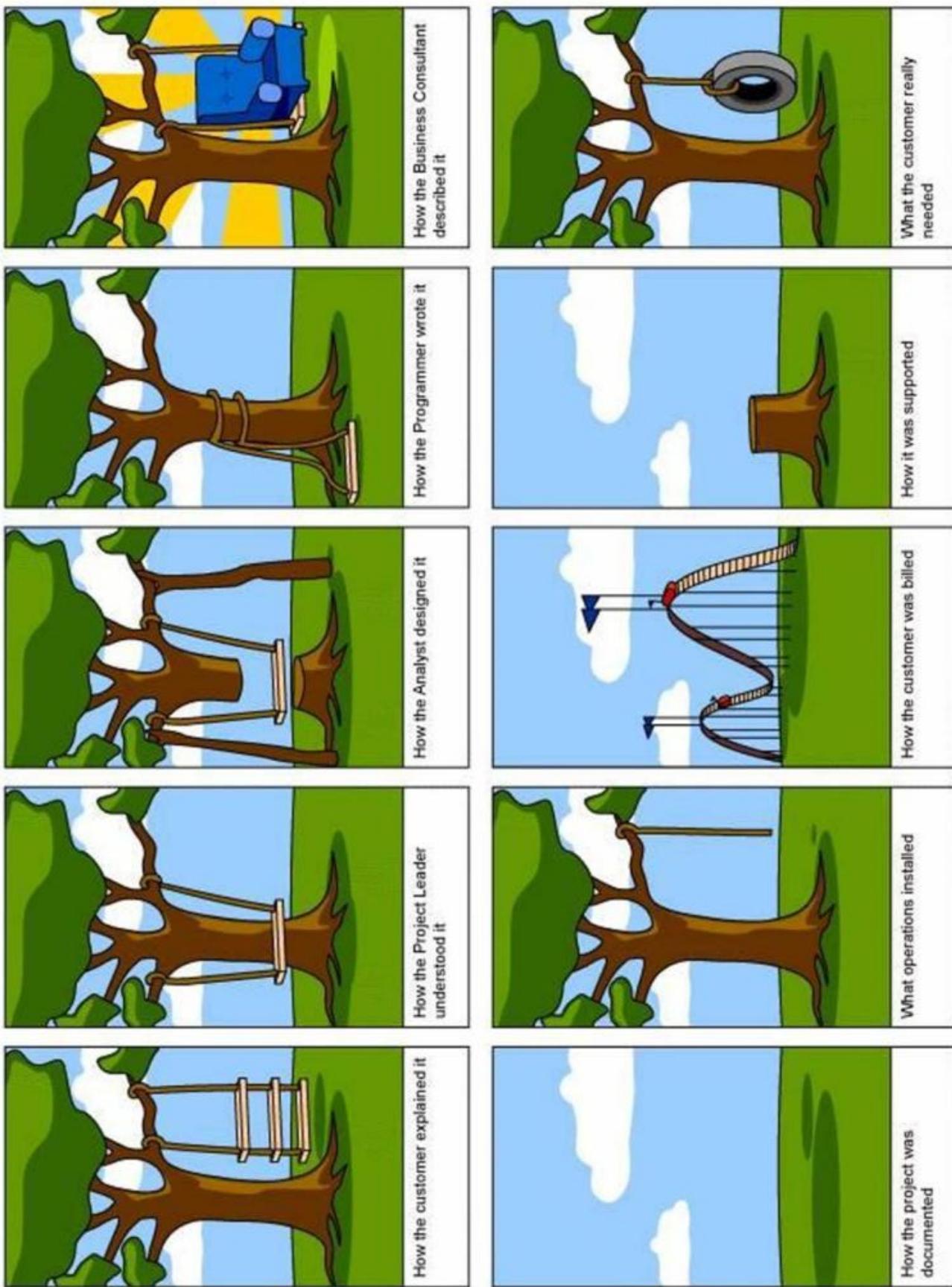
modelos en espiral (spiral models) :



modelos de desarrollo ágil (agile development models) como la **programación extrema (extreme programming)** :



Estos conceptos se analizan con detalle en la asignatura **Ingeniería de Software (Software Engineering)**



Tema VII

Recursividad

Objetivos:

- ▷ Entender el concepto de recursividad.
- ▷ Conocer los fundamentos del diseño de algoritmos recursivos.
- ▷ Comprender la ejecución de algoritmos recursivos.
- ▷ Aprender a realizar trazas de algoritmos recursivos.
- ▷ Comprender las ventajas e inconvenientes de la recursividad.

VII.1. El concepto matemático de Recursividad

VII.1.1. Soluciones recursivas

"La iteración es humana; la recursividad, divina".



"Para entender la recursividad es necesario antes entender la recursividad".

Recursividad (recursion) es la forma en la cual se especifica un proceso basado en su propia definición.

Podemos usar recursividad si la solución de un problema está expresada en términos de la resolución del mismo tipo de problema, aunque de **menor tamaño** y conocemos la solución no-recursiva para un determinado **caso base (base case)** (o varios).

En Matemáticas se usa este concepto en varios ámbitos:

- ▷ Demostración por **inducción (induction)** :
 - Demostrar para un caso base.
 - Demostrar para un tamaño n , considerando que está demostrado para un tamaño menor que n .
- ▷ Definición recursiva

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n - 1)! & \text{si } n > 0 \end{cases}$$

- ▷ No es necesario definir la secuencia de pasos exacta para resolver el problema. Podemos considerar que “*tenemos resuelto*” el problema (de menor tamaño).
- ▷ Usaremos recursividad cuando surja de forma natural al plantear la solución a un problema. Los problemas más interesantes se verán en el segundo cuatrimestre.

Ejemplo. Cálculo del factorial con n=3.

$$3! = 3 * 2!$$

(1)

$$3! = 3 * \boxed{2!}$$

↓

$$2! = 2 * 1!$$

(2)

$$3! = 3 * \boxed{2!}$$

↓

$$2! = 2 * \boxed{1!}$$

↓

$$1! = 1 * 0!$$

(3)

$$3! = 3 * \boxed{2!}$$

↓

$$2! = 2 * \boxed{1!}$$

↓

$$1! = 1 * \boxed{0!}$$

↓

$$0! = 1$$

(CASO BASE)

(4)

$$\begin{array}{c}
 3! = 3 * \boxed{2!} \\
 \downarrow \\
 2! = 2 * \boxed{1!} \\
 \downarrow \\
 1! = 1 * \circled{1}^{(0!)} \\
 \downarrow \\
 1
 \end{array}$$

(5)

$$\begin{array}{c}
 3! = 3 * \boxed{2!} \\
 \downarrow \\
 2! = 2 * \circled{1}^{(1!)} \\
 \downarrow \\
 1! = 1 * 1 = 1
 \end{array}$$

(6)

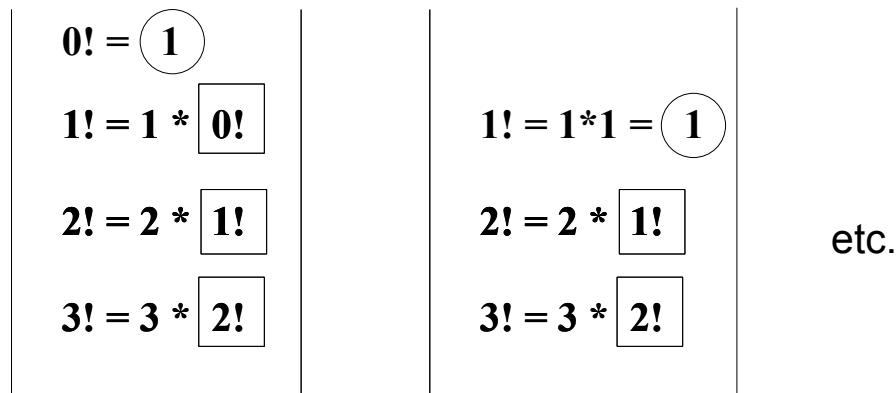
$$\begin{array}{c}
 3! = 3 * \circled{2}^{(2!)} \\
 \downarrow \\
 2! = 2 * 1 = 2
 \end{array}$$

(7)

$$3! = 3 * 2 = 6$$

(8)

En resumen:



VII.1.2. Diseño de algoritmos recursivos

Debemos diseñar:

▷ **Casos base:**

Son los casos del problema que se resuelve con un segmento de código sin recursividad.

Para que una definición recursiva esté completamente identificada es necesario tener un caso base que no se calcule utilizando casos anteriores y que la división del problema converja a ese caso base.

Siempre debe existir al menos un caso base

El número y forma de los casos base son hasta cierto punto arbitrarios. La solución será mejor cuanto más simple y eficiente resulte el conjunto de casos seleccionados.

$$0! = 1$$

▷ **Casos generales:**

Debemos encontrar la solución a un problema (o varios) del mismo tipo, pero de menor tamaño.

$$(n-1)!$$

▷ **Composición:**

Una vez encontradas las soluciones a los problemas menores, habrá que ejecutar un conjunto de pasos adicionales. Estos pasos, junto con las soluciones a los subproblemas, componen la solución al problema general que queremos resolver.

$$n! = n * (n-1)!$$

Ejercicio. Plantear la solución recursiva al problema de calcular la potencia de un real x elevado a un entero positivo n , es decir: x^n

▷ **Expresado con palabras:**

– **Caso base:** Solución de Potencia($x, 0$) = 1

– **Caso general:**

Para hallar la solución de Potencia(x, n),

hallar primero la solución de Potencia($x, n - 1$)

– **Composición:**

Solución de Potencia(x, n) =

$x * \text{Solución de Potencia}(x, n - 1)$

▷ **Expresado en lenguaje matemático:**

– **Caso base:** $x^0 = 1$

– **Caso general:** x^{n-1}

– **Composición:** $x^n = x * x^{n-1}$

Ejercicio. Plantear la solución recursiva al problema de sumar los enteros positivos menores que un entero n , es decir: $\sum_{i=1}^{i=n} i$

▷ **Expresado con palabras:**

– **Caso base:** Solución de SumaHasta(1) = 1

– **Caso general:**

Para hallar la solución de SumaHasta(n),

hallar primero la solución de SumaHasta($n - 1$),

– **Composición:**

Solución de SumaHasta(n) =

$n +$ Solución de SumaHasta($n - 1$)

▷ **Expresado en lenguaje matemático:**

– **Caso base:** $\sum_{i=1}^{i=1} i = 1$

– **Caso general:** $\sum_{i=1}^{i=n-1} i$

– **Composición:** $\sum_{i=1}^{i=n} i = n + \sum_{i=1}^{i=n-1} i$

Ejercicio. Plantear la solución recursiva al problema de multiplicar dos enteros a y b , es decir: $a * b$

▷ **Expresado con palabras:**

– **Caso base:** Solución de Multiplica cualquier número con $0 = 0$

– **Caso general:**

Para hallar la solución de Multiplica(a, b),

hallar primero la solución de Multiplica($a, b - 1$)

– **Composición:**

Solución de Multiplica(a, b) =

$a +$ Solución de Multiplica($a, b - 1$)

▷ **Expresado en lenguaje matemático:**

– **Casos base:** $0 * b = 0, \ a * 0 = 0$

– **Caso general:** $a * (b - 1)$

– **Composición:** $a * b = a + a * (b - 1)$

VII.2. Funciones recursivas

VII.2.1. Definición de funciones recursivas

¿Cómo se expresa en un lenguaje de programación la solución a un problema? Una solución será el valor devuelto por la llamada a una función.

$$n! \rightarrow \text{Factorial}(n)$$

¿Cómo se traduce a un lenguaje de programación el planteamiento recursivo de una solución? A través de una función que, dentro del código que la define, realiza una o más llamadas a la propia función. Una función que se llama a sí misma se denomina *función recursiva (recursive function)*.

```
\\" Prec: 0 <= n <= 20
long long Factorial (int n) {
    long long resultado;

    if (n==0)
        resultado = 1;                      //Caso base
    else
        resultado = n*Factorial(n-1); //Caso general

    return resultado;
}

\\" Prec: 0 <= n <= 20
long long Factorial (int n) {
    if (n==0)
        return 1;                          //Caso base
    else
        return n*Factorial(n-1);          //Caso general
}
```

VII.2.2. Ejecución de funciones recursivas

En general, en la pila se almacena el marco asociado a las distintas funciones que se van activando. En particular, cada llamada recursiva genera una nueva zona de memoria en la pila independiente del resto de llamadas.

Ejemplo. Ejecución del factorial con n=3.

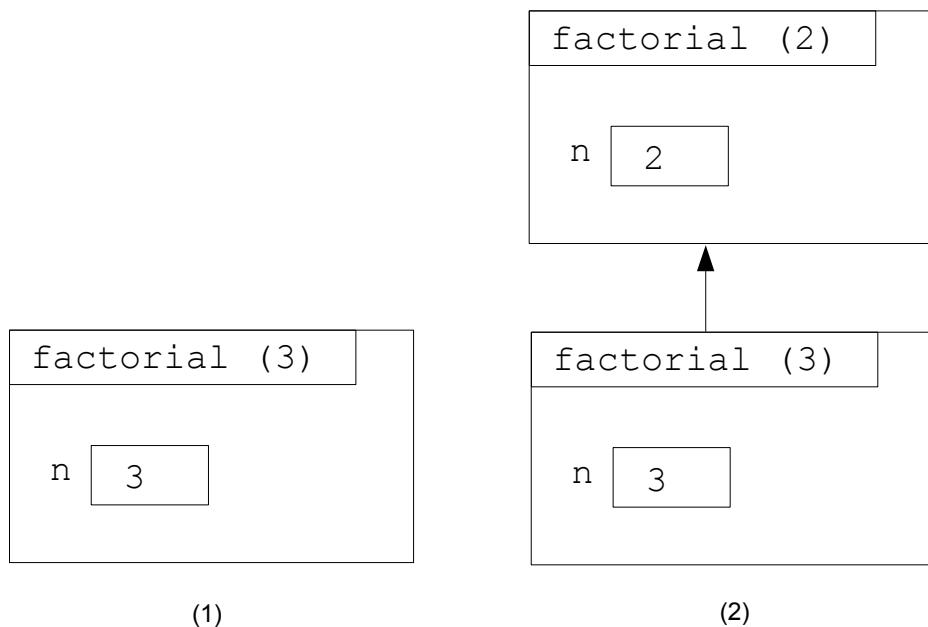
1. Dentro de factorial, cada llamada

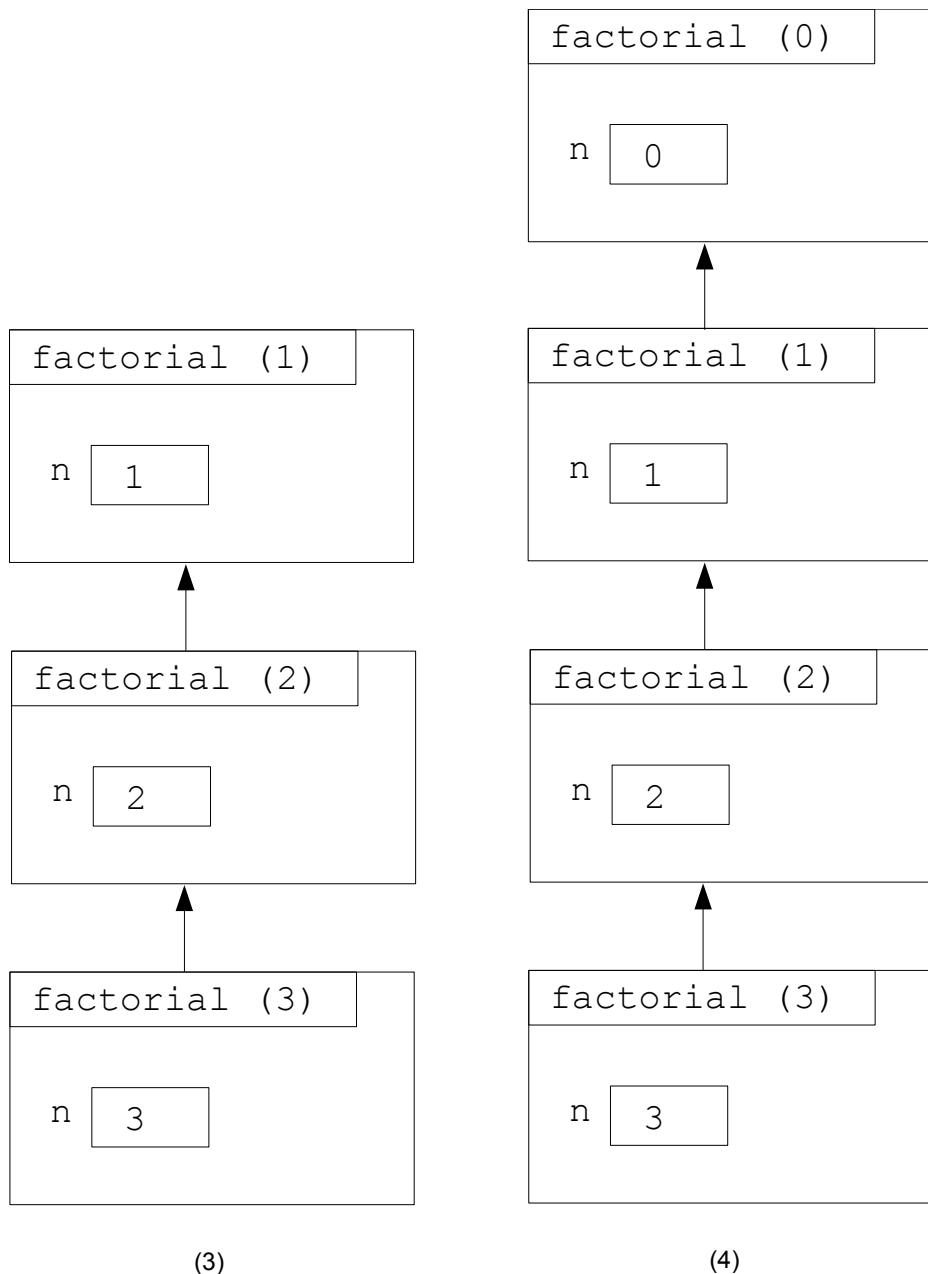
```
return (n * factorial(n-1));
```

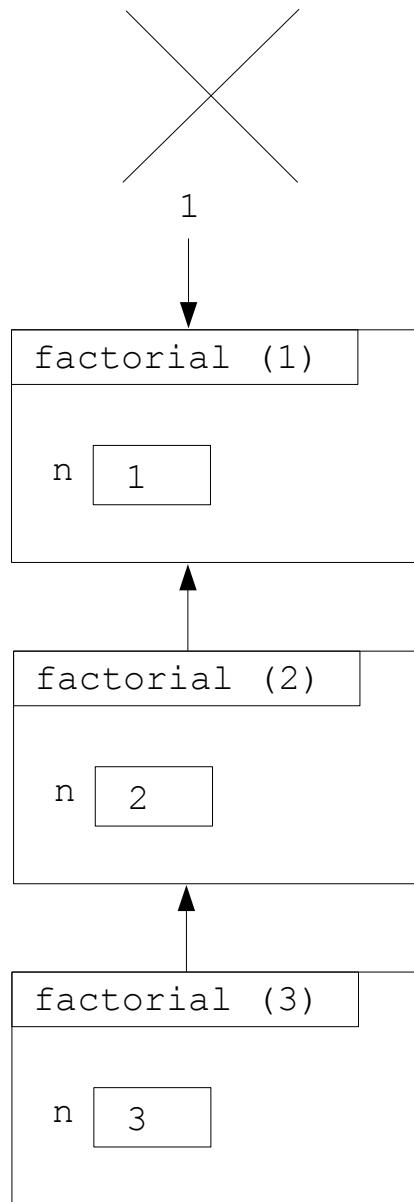
genera una nueva zona de memoria en la pila, siendo $n-1$ el correspondiente parámetro actual para esta zona de memoria y queda pendiente la evaluación de la expresión y la ejecución del `return`.

2. El proceso anterior se repite hasta que la condición del caso base se hace cierta.

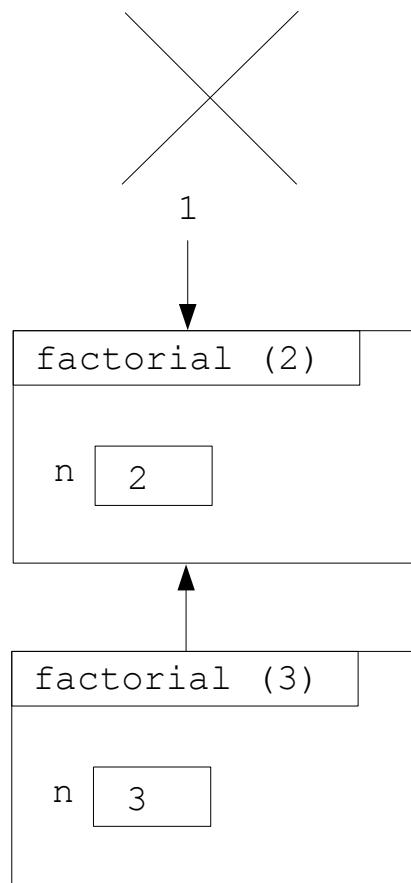
- ▷ Se ejecuta la sentencia `return 1;`
- ▷ Empieza la vuelta atrás de la recursión, se evalúan las expresiones y se ejecutan los `return` que estaban pendientes.



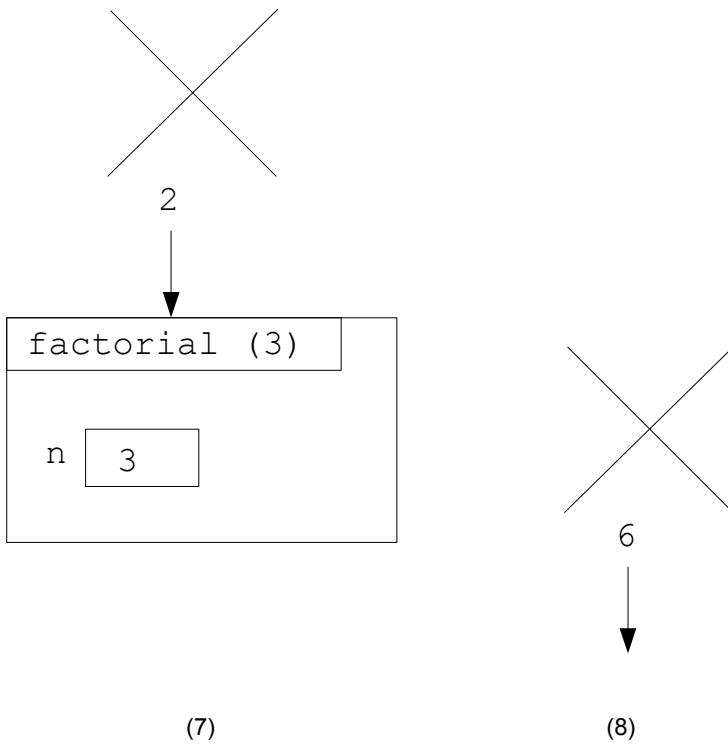




(5)



(6)

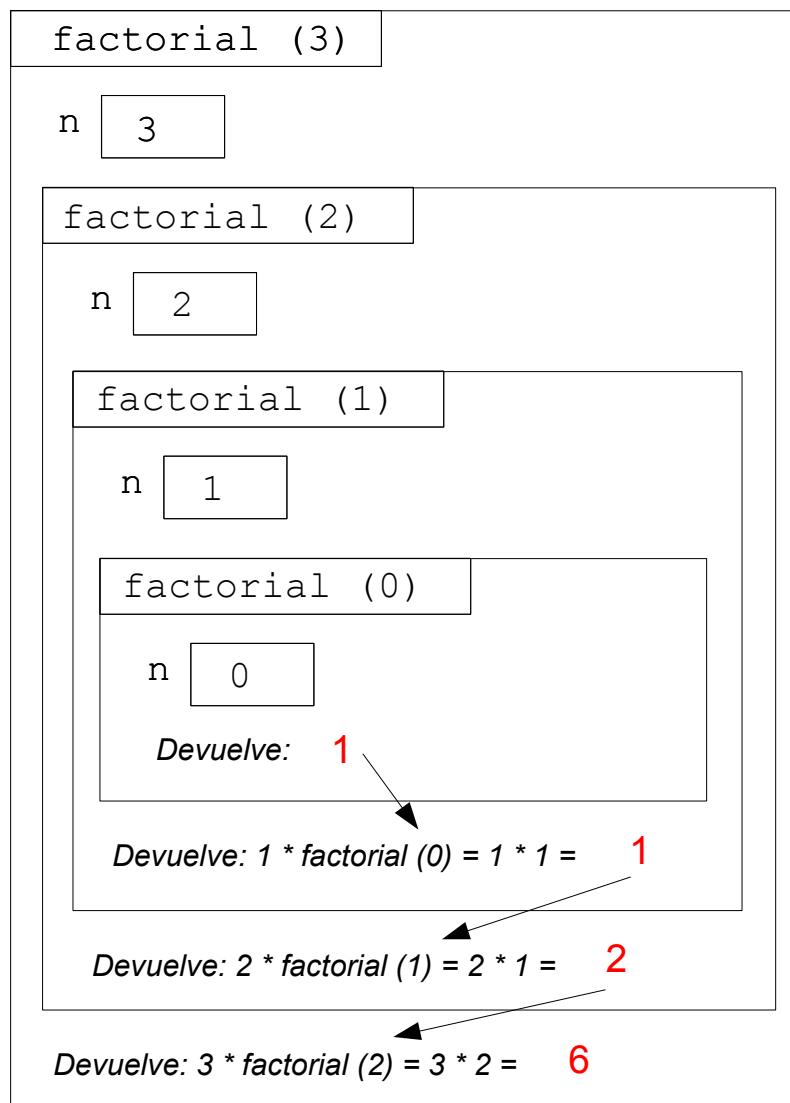


Cada llamada a una función recursiva genera un marco de trabajo en la pila → Cada marco tiene su propia copia del parámetro formal.

Cada llamada a una función recursiva genera un marco de trabajo en la pila → Esto supone una recarga computacional importante

Una forma alternativa de representar gráficamente las llamadas recursivas es hacerlo en cascada:

Llamada: factorial(3)



Ejercicio. Calcular la potencia de un real elevado a un entero positivo n

- ▷ **Caso base ($n = 0$):** $x^0 = 1$
- ▷ **Caso general:** x^{n-1}
- ▷ **Composición:** $x^n = x * x^{n-1}$

```
// Prec: n >= 0
double Potencia(double x, int n){
    if (n == 0)
        return 1.0;
    else
        return x * Potencia(x, n - 1);
}
```

Potencia(3.0, 300) **devuelve** 3.63603 e+238

Potencia(3.0, 700) **devuelve** 1.#INF

Potencia(3.0, 800) **devuelve** 1.#INF

Potencia(3.0, 2000) **produce un error en tiempo de ejecución.**

Como cada llamada recursiva implica la creación de un marco de trabajo en la pila, la memoria asociada al proceso del programa puede saturarse, produciéndose entonces un *desbordamiento de la pila (stack overflow)* y por tanto, un error en tiempo de ejecución.

Ejercicio. Sumar los enteros positivos menores que un entero n

- ▷ **Caso base** ($n = 1$): $\sum_{i=1}^{i=1} i = 1$
- ▷ **Caso general:** $\sum_{i=1}^{i=n-1} i$
- ▷ **Composición:** $\sum_{i=1}^{i=n} i = n + \sum_{i=1}^{i=n-1} i$

```
\\" Prec: n > 0
long long SumaPositivosHasta(int n){
    if (n <= 1)
        return 1;
    else
        return n + SumaPositivosHasta(n - 1);
}
```

Si se viola la precondición

```
total = SumaPositivosHasta(-5);
```

el resultado sería un valor erróneo. Si optamos por suprimir la precondición, entonces debemos añadir más casos base:

```
long long SumaPositivosHasta(int n){
    if (n <= 0)
        return -1; // o mejor, lanzar una excepción
    else if (n == 1)
        return 1;
    else
        return n + SumaPositivosHasta(n - 1);
}
```

Supongamos la llamada siguiente:

```
int main(){
    long long suma_total;

    suma_total = SumaPositivosHasta(10);
```

¿Cuántas veces se comprueban las condiciones?:

- ▷ if (n <= 0): 11 veces.
- ▷ if (n == 1): 11 veces

Los condicionales incluidos en el código de una función recursiva se evalúan cada vez que hay una llamada recursiva.

Si suponemos que las llamadas se realizarán usualmente con valores positivos (10, 4, 8, etc), podríamos mejorar la eficiencia cambiando el orden de comprobación:

```
long long SumaPositivosHasta(int n){
    if (n > 1)
        return n + SumaPositivosHasta(n - 1);
    else if (n == 1)
        return 1;
    else
        return -1; // o mejor, lanzar excepción
}
```

De todas formas, como ya hemos señalado anteriormente, la recarga computacional importante es debida a la creación de un marco en la pila por cada llamada recursiva.

Ejercicio. Multiplicar dos enteros positivos $a * b$

- ▷ **Casos base:** $0 * b = 0$, $a * 0 = 0$
- ▷ **Caso general:** $a * (b - 1)$
- ▷ **Composición:** $a * b = a + a * (b - 1)$

```
long long Producto(unsigned int a, unsigned int b){
    if (b == 0 || a == 0)
        return 0;
    else
        return a + producto(a, b-1);
}
```

El caso base correspondiente al condicional $a == 0$ no sería estrictamente necesario: en las llamadas recursivas iría sumando 0 y también funcionaría correctamente. En definitiva:

```
long long Producto(unsigned int a, unsigned int b){
    if (b == 0 || a == 0)           // Realiza siempre más comprobaciones
        return 0;
    else
        return a + producto(a, b-1);
}
```

```
long long Producto(unsigned int a, unsigned int b){
    if (b == 0)                   // Realiza más llamadas cuando a es 0
        return 0;
    else
        return a + producto(a, b-1);
}
```

VII.3. Clases con métodos recursivos

VII.3.1. Métodos recursivos

El planteamiento y definición de un método recursivo es análogo a las funciones. La diferencia es que todas las llamadas recursivas del método pueden acceder a los datos miembro del objeto. Si una llamada recursiva modifica algún dato miembro, el resto de llamadas se verán afectadas, ya que todas ellas acceden a los mismos datos miembro.

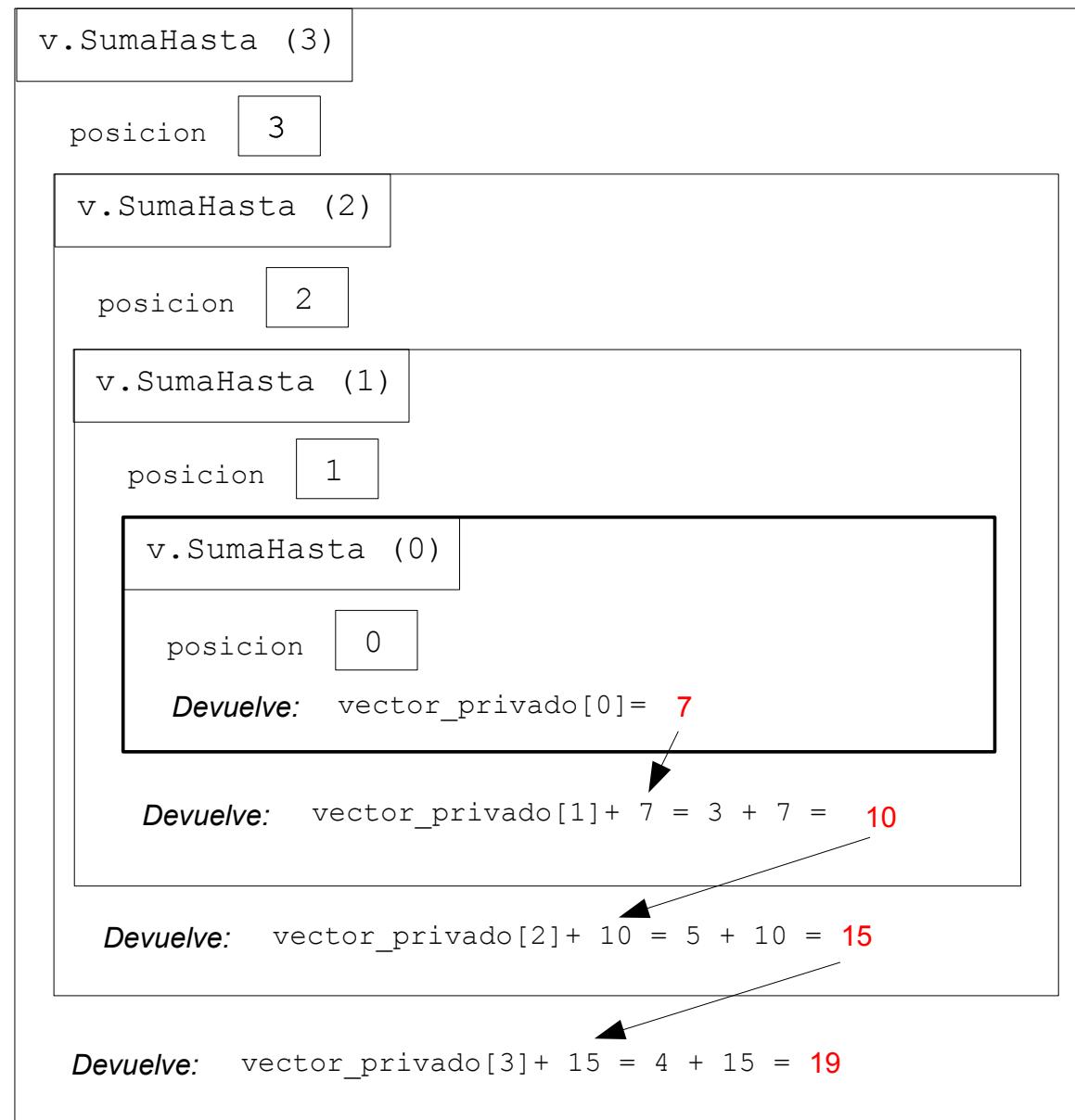
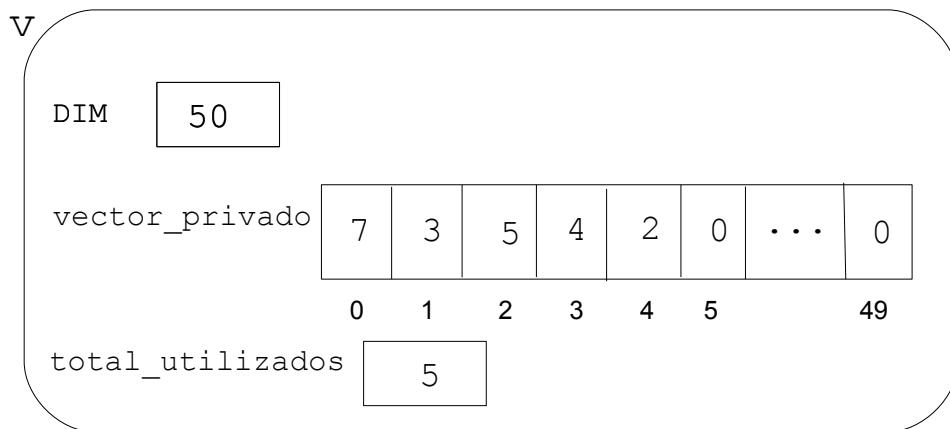
Cada llamada a un método recursivo genera un marco de trabajo en la pila. Cada marco tiene su propia copia del parámetro formal. Sin embargo, todos los marcos acceden a los mismos datos miembro.

Ejercicio. Plantear una solución recursiva al problema de sumar los elementos de un vector y definir un método recursivo que la implemente.

```
class MiVectorEnteros{  
private:  
    static const int DIM = 50;  
    int vector_privado[DIM];  
    int total_utilizados;  
public:  
    .....  
    long long SumaHasta(int posicion){  
        .....  
    }  
}
```

Cada llamada recursiva tendrá su propio valor de `posicion`, pero todas las llamadas accederán al mismo dato miembro `vector_privado`

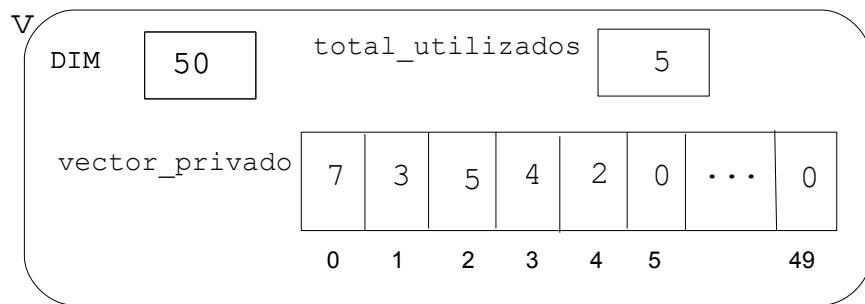
```
class MiVectorEnteros{  
private:  
    static const int DIM = 50;  
    int vector_privado[DIM];  
    int total_utilizados;  
public:  
    MiVectorEnteros()  
        :total_utilizados(0)  
    {  
    }  
    int TotalUtilizados(){  
        return total_utilizados;  
    }  
    void Aniade(int nuevo){  
        if (total_utilizados < DIM){  
            vector_privado[total_utilizados] = nuevo;  
            total_utilizados++;  
        }  
        // else  
        //     lanzar excepción  
    }  
    int Elemento(int indice){  
        return vector_privado[indice];  
    }  
    long long SumaHasta(int posicion){  
        if (posicion > 0 && posicion < total_utilizados)  
            return vector_privado[posicion] + SumaHasta(posicion-1);  
        else if (posicion == 0)  
            return vector_privado[0];  
        // else  
        //     lanzar excepción  
    }  
};
```



Ejercicio. Plantear la solución recursiva al problema de buscar la primera ocurrencia de una componente de un vector y definir un método que la implemente.

Observad que, en este ejemplo, se necesitan dos casos base.

```
class MiVectorEnteros{  
private:  
    static const int DIM = 50;  
    int vector_privado[DIM];  
    int total_utilizados;  
public:  
    .....  
    // Prec: 0 < izda , dcha < TotalUtilizados()  
    int PrimeraOcurrenciaEntre(int izda, int dcha, int buscado){  
        if (izda > dcha)  
            return -1;  
        else if (vector_privado[izda] == buscado)  
            return izda;  
        else  
            return PrimeraOcurrenciaEntre(izda + 1, dcha, buscado);  
    }  
};
```



v.PrimeraOcurrenciaEntre (0, 4, 5)

izda 0 dcha 4 buscado 5
 izda < dcha vector_privado[0] != 5

v.PrimeraOcurrenciaEntre (1, 4, 5)

izda 1 dcha 4 buscado 5
 izda < dcha vector_privado[1] != 5

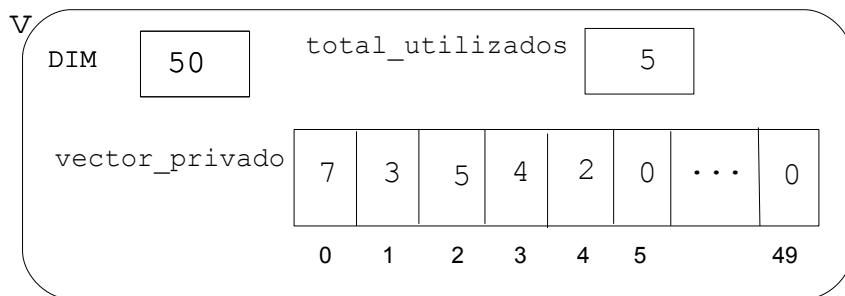
v.PrimeraOcurrenciaEntre (2, 4, 5)

izda 2 dcha 4 buscado 5
 izda < dcha vector_privado[2] == 5

Devuelve: 2

Devuelve: 2

Devuelve: 2



v.PrimeraOcurrenciaEntre (0, 3, 6)

izda 0 dcha 3 buscado 6

izda < dcha vector_privado[0] != 6

v.PrimeraOcurrenciaEntre (1, 3, 6)

izda 1 dcha 3 buscado 6

izda < dcha vector_privado[1] != 6

v.PrimeraOcurrenciaEntre (2, 3, 6)

izda 2 dcha 3 buscado 6

izda < dcha vector_privado[1] != 6

v.PrimeraOcurrenciaEntre (3, 3, 6)

izda 3 dcha 3 buscado 6

izda == dcha vector_privado[2] != 6

v.PrimeraOcurrenciaEntre (4, 3, 6)

izda 4 dcha 3 buscado 6

izda > dcha

Devuelve: -1

Devuelve: -1

Devuelve: -1

Devuelve: -1

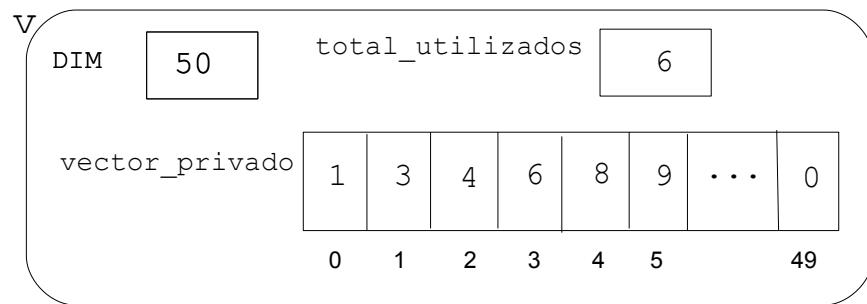
Devuelve: -1

Ejercicio. Plantear la solución recursiva al problema de buscar con el método de **búsqueda binaria** la primera ocurrencia de una componente de un vector ordenado y definir un método que la implemente.

Recordemos el método de búsqueda binaria:

```
class MiVectorEnteros{  
private:  
    static const int DIM = 50;  
    int vector_privado[DIM];  
    int total_utilizados;  
public:  
    .....  
    int BusquedaBinaria (int buscado){  
        int izda, dcha, centro;  
  
        izda = 0;  
        dcha = total_utilizados-1;  
        centro = (izda+dcha)/2;  
  
        while (izda <= dcha && vector_privado[centro] != buscado) {  
            if (buscado < vector_privado[centro])  
                dcha = centro-1;  
            else  
                izda = centro+1;  
  
            centro = (izda+dcha)/2;  
        }  
  
        if (izda > dcha)  
            return -1;  
        else  
            return centro;  
    }  
};
```

```
class MiVectorEnteros{  
private:  
    static const int DIM = 50;  
    int vector_privado[DIM];  
    int total_utilizados;  
public:  
    .....  
    // Prec: 0 < izda , dcha < TotalUtilizados()  
    int BusquedaBinaria(int izda, int dcha, int buscado){  
        int centro;  
  
        if (izda > dcha)  
            return -1;  
        else{  
            centro = (izda + dcha) / 2;  
  
            if (vector_privado[centro] == buscado)  
                return centro;  
            else  
                if (vector_privado[centro] > buscado)  
                    return BusquedaBinaria(izda, centro-1, buscado);  
                else  
                    return BusquedaBinaria(centro + 1, dcha, buscado);  
        }  
    }  
};
```



v.BusquedaBinaria (0, 5, 6)

izda [0] dcha [5] buscado [6] centro [2]

vector_privado[2] < 6

v.BusquedaBinaria (3, 5, 6)

izda [3] dcha [5] buscado [6] centro [4]

vector_privado[4] > 6

v.BusquedaBinaria (3, 3, 6)

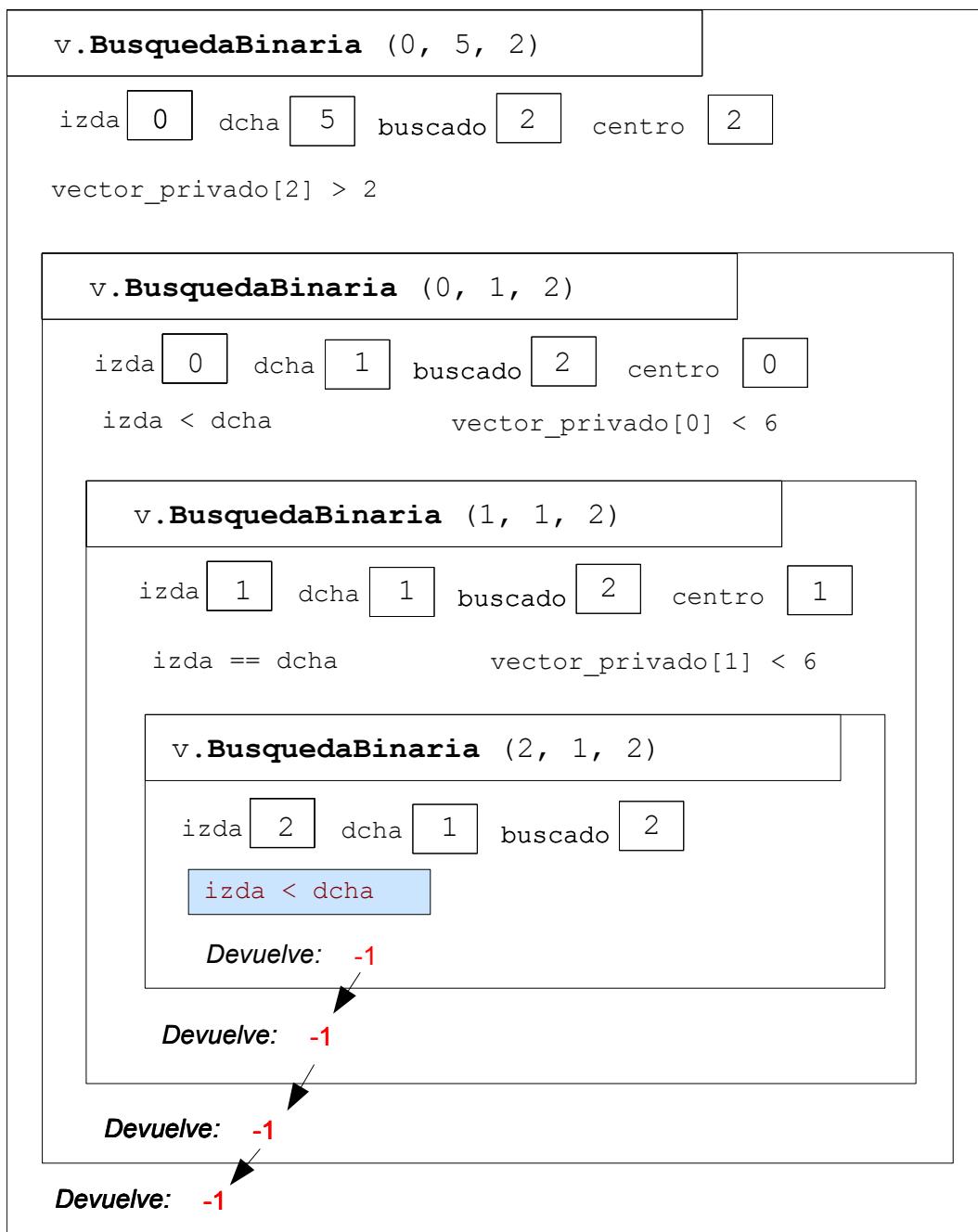
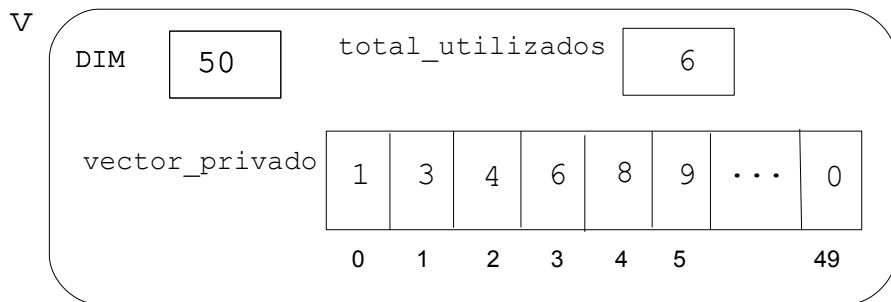
izda [3] dcha [3] buscado [6] centro [3]

vector_privado[3] == 6

Devuelve: 3

Devuelve: 3

Devuelve: 3



Ejercicio. Plantear la solución recursiva al problema de encontrar el mayor elemento de un vector (entre dos posiciones dadas) y definir un método que la implemente.

```
class MiVectorEnteros{  
private:  
    static const int DIM = 50;  
    int vector_privado[DIM];  
    int total_utilizados;  
public:  
    .....  
    // Prec: 0 < izda , dcha < TotalUtilizados()  
    int PosMayor(int izda, int dcha){  
        int pos_mayor_anterior;  
  
        if (izda < dcha){  
            pos_mayor_anterior = PosMayor(izda + 1, dcha);  
  
            if (vector_privado[pos_mayor_anterior] > vector_privado[izda])  
                return pos_mayor_anterior;  
            else  
                return izda;  
        }  
        else if (izda == dcha)  
            return izda;  
        else  
            return -1;  
    }  
};
```

Ejemplo. Invertir un vector (el subvector entre las posiciones izda y dcha)

Si izda == dcha

Solución: Construir un vector con esa única componente

si no

Calcular la solución al problema de invertir el vector entre las posiciones izda + 1 y dcha

Solución: Añadir la componente izda al final de la solución anterior

```
class MiVectorCaracteres{  
private:  
    static const int DIM = 50;  
    char vector_privado[DIM];  
    int total_utilizados;  
public:  
    ....  
    // Prec: 0 < izda , dcha < TotalUtilizados()  
  
    MiVectorCaracteres InvierteRecursivo(int izda, int dcha){  
        MiVectorCaracteres inverso;  
  
        if (izda < dcha){  
            inverso = InvierteRecursivo(izda+1, dcha);  
            inverso.Anade(vector_privado[izda]);  
        }  
        else if (izda == dcha)  
            inverso.Anade(vector_privado[izda]);  
  
        return inverso;      // Si izda > dcha, devuelve vector vacío.  
    }  
};
```

VII.3.2. Ordenación con Quicksort (Ampliación)

Ideado por Charles Antony Richard Hoare en 1960.



Este es un método de ordenación que utiliza la recursividad para ir fijando los rangos del vector sobre los que otro método (*partir*) realizará los intercambios necesarios para ir ordenándolo. *Idea*:

- ▷ Para ordenar alfabéticamente un montón de exámenes, se hacen dos pilas: en una se van echando los que tienen apellidos menores que un *pivote*, por ejemplo, 'M' y en la otra los mayores que dicho pivote.
- ▷ Se vuelve a repetir el proceso con cada montón, pero cambiando el elemento *pivote*. En la segunda iteración recursiva, en el primer montón podría escogerse como pivote 'J' y en el segundo 'R'.
- ▷ Si no conocemos a priori el rango de valores, el pivote lo elegimos de forma arbitraria (por ejemplo, el primero que cojamos de la pila)

```
class MiVectorEnteros{  
private:  
    int partir(int primero, int ultimo)                // <- No recursivo  
    .....  
public:  
    void QuickSort(int inicio, int final){           // <- Recursivo  
        int pos_pivote;  
  
        if (inicio < final) {  
            pos_pivote = partir (inicio, final);  
            QuickSort (inicio, pos_pivote - 1);        // Ordena primera mitad  
            QuickSort (pos_pivote + 1, final);          // Ordena segunda mitad  
        }  
    }  
}
```

Líneas básicas del algoritmo de la función partir:

1. Tomar un elemento arbitrario del vector: **pivote**
2. Recorrer el vector de izquierda a derecha hasta encontrar un elemento situado en una posición **izda** tal que $v[izda] > \text{pivote}$
3. Recorrer el vector de derecha a izquierda hasta encontrar otro elemento situado en una posición **dcha** tal que $v[dcha] < \text{pivote}$
4. Intercambiar los elementos de las casillas **izda** y **dcha**

Una vez hecho el intercambio tendremos que:

$$v[izda] < \text{pivote} < v[dcha]$$

5. Repetir hasta que los dos procesos de recorrido se encuentren ($\text{izda} > \text{dcha}$).
6. Colocar el pivote en el sitio que le corresponde.
7. Una vez hecho lo anterior, el vector está particionado en dos zonas delimitadas por el pivote. El pivote está ya colocado correctamente en su sitio.

	0	1	2	3	4	5	6	7	8	9
(A)	4	2	5	2	6	10	3	7	6	4

pivote

	0	1	2	3	4	5	6	7	8	9
(B)	4	2	5	2	6	10	3	7	6	4

izda**dcha**

	0	1	2	3	4	5	6	7	8	9
(C)	4	2	3	2	6	10	5	7	6	4

	0	1	2	3	4	5	6	7	8	9
(D)	4	2	3	2	6	10	5	7	6	4

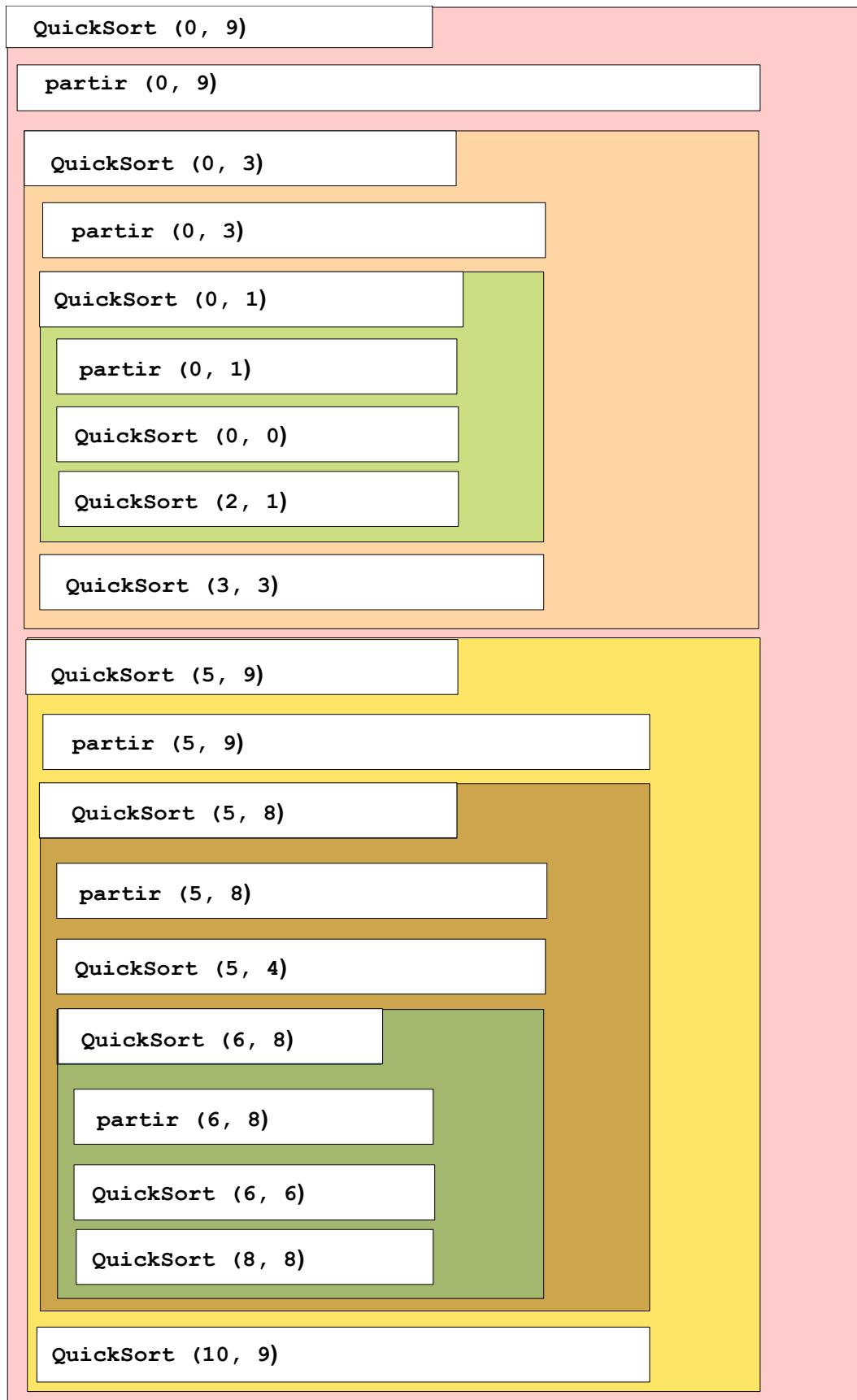
dcha izda

	0	1	2	3	4	5	6	7	8	9
(E)	4	2	3	2	6	10	5	7	6	4

	0	1	2	3	4	5	6	7	8	9
(F)	4	2	3	2	6	10	5	7	6	4

	0	1	2	3	4	5	6	7	8	9
(G)	2	2	3	4	6	10	5	7	6	4

```
int partir (int primero, int ultimo){  
    int intercambia;  
    int izda, dcha;  
    int valor_pivote = vector_privado[primero];  
  
    izda = primero + 1;      // izda avanza hacia delante  
    dcha = ultimo;          // dcha retrocede hacia atrás  
  
    while (izda <= dcha){  
        while (izda <= dcha && vector_privado[izda] <= valor_pivote)  
            izda++;  
  
        while (izda <= dcha && vector_privado[dcha] >= valor_pivote)  
            dcha--;  
  
        if (izda < dcha) {  
            intercambia = vector_privado[izda];  
            vector_privado[izda] = vector_privado[dcha];  
            vector_privado[dcha] = intercambia;  
            dcha--;  
            izda++;  
        }  
    }  
    intercambia = vector_privado[primero];  
    vector_privado[primero] = vector_privado[dcha];  
    vector_privado[dcha] = intercambia;  
  
    return dcha;  
}
```



Ampliación:

- ▷ Quicksort es uno de los mejores algoritmos de ordenación. Se analizará su eficiencia en el segundo cuatrimestre.
- ▷ Quicksort puede parallelizarse fácilmente. Cada CPU se encargaría de la resolución de una llamada recursiva. Al final basta juntar los subvectores ordenados. Esto es posible porque el pivot calculado en cada momento, es colocado en el lugar que le corresponde.
- ▷ La elección del pivot es muy importante. Las mejoras de Quicksort pasan por desarrollar mecanismos rápidos de elección de *buenos* pivotes (elementos que dejen más o menos el mismo número de elementos a la izquierda y derecha)
- ▷ Algunos applets de demostración.

Para tener una idea *global*:

<http://maven.smith.edu/~thiebaut/java/sort/>

Ejecución paso a paso:

<http://pages.stern.nyu.edu/~panos/java/Quicksort/>

<http://math.hws.edu/TMCM/java/xSortLab/> (en este applet, partir recorre primero la derecha y luego la izquierda)

VII.4. Recursividad versus iteración

Desventajas de la recursividad:

- ▷ La carga computacional (tiempo-espacio) asociada a una llamada a una función y el retorno a la función que hace la llamada.
- ▷ Algunas soluciones recursivas pueden hacer que la solución para un determinado tamaño del problema se calcule varias veces.

Ventajas de la recursividad:

- ▷ La solución recursiva suele ser concisa, legible y elegante.
Tal es el caso, por ejemplo, del recorrido de estructuras *complejas* como árboles, listas, grafos, etc. Se verá en el segundo cuatrimestre.

Habrá que analizar en cada caso la conveniencia o no de usar recursividad.

Recursividad de cola

Se dice que una función tiene *recursividad de cola (tail recursion)* si sólo se produce una llamada recursiva y no se ejecuta posteriormente ningún código. Estas funciones pueden traducirse fácilmente a un algoritmo no recursivo, más eficiente. De hecho, muchos compiladores realizan automáticamente esta conversión.

Los métodos BusquedaBinaria y PrimeraOcurrenciaEntre tienen recursividad de cola. En el resto de ejemplos vistos, siempre hay que hacer algo más después de realizar la llamada recursiva. Por ejemplo:

```
// Prec: exponente >= 0
double Potencia(double base, int exponente){
    if (exponente == 0)
        return 1.0;
    else
        return base * Potencia(base, exponente-1);
}
```

En cualquier caso, muchos métodos/funciones sin recursividad de cola también pueden traducirse de forma fácil como un algoritmo iterativo.

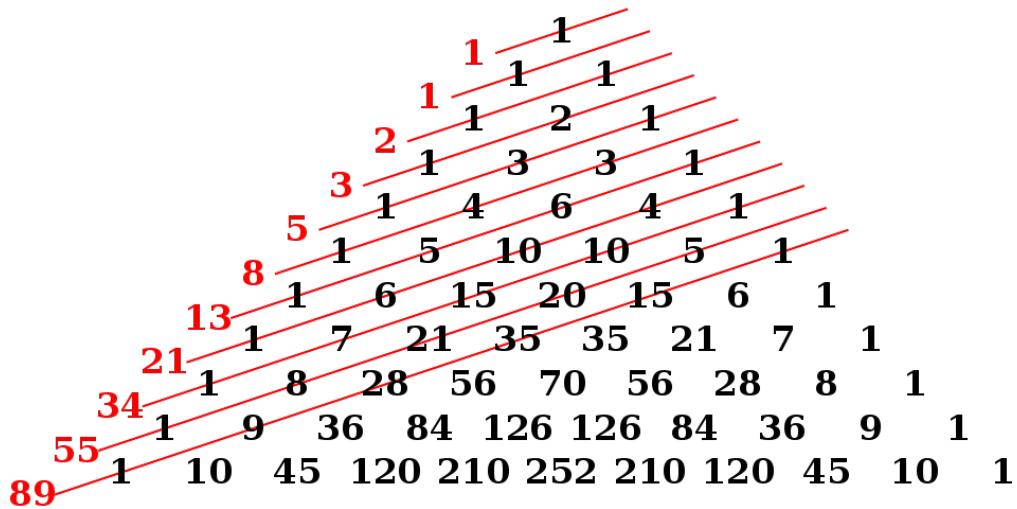
Repetición de cómputos

Los algoritmos *interesantes* serán los que realicen más de una llamada recursiva (como Quicksort), pero debemos evitar la repetición de cómputos.

Ejemplo. Sucesión de Fibonacci.

$$Fib(0) = Fib(1) = 1$$

$$Fib(n) = Fib(n - 1) + Fib(n - 2)$$



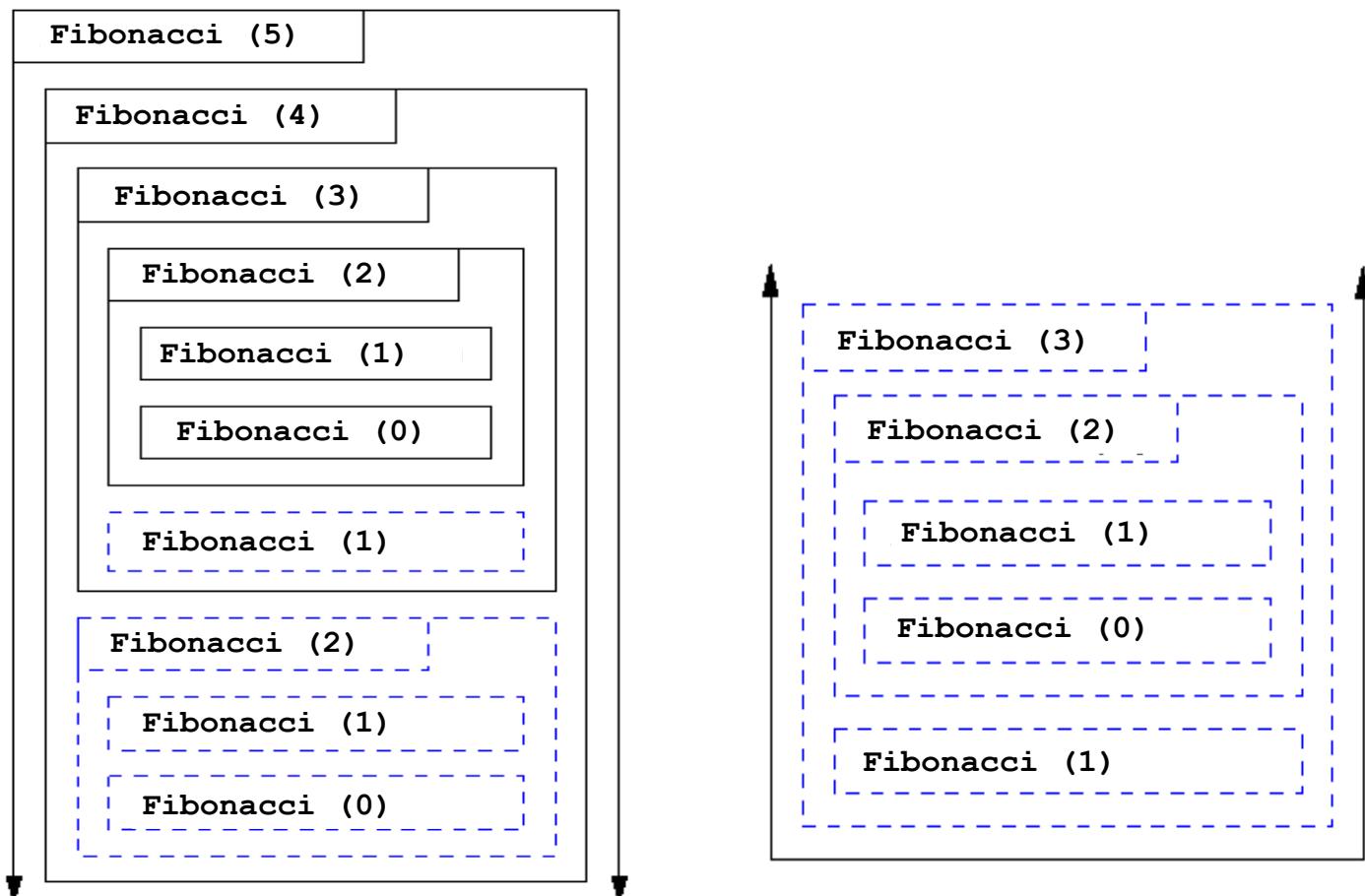
Ampliación:



La sucesión de Fibonacci tiene muchas aplicaciones. Por ejemplo $Fibonacci(n+2)$ da como resultado el número de subconjuntos de $\{1, \dots, n\}$ que no contienen enteros consecutivos. En la naturaleza, están presentes en la fórmula que determina cómo se disponen las hojas en los tallos. Hay una revista matemática dedicada exclusivamente a resultados sobre esta sucesión. Para más información, consultad wikipedia (en inglés) http://en.wikipedia.org/wiki/Fibonacci_number y la web de la Enciclopedia de sucesiones enteras <http://oeis.org/A000045>.

```
long long Fibonacci (int n){  
    if (n == 0 || n == 1)  
        return 1;  
    else  
        return Fibonacci(n-1) + Fibonacci(n-2);  
}
```

Cálculo de Fibonacci(5):



La solución recursiva de Fibonacci repite muchos cómputos, por lo que es muy ineficiente.

La solución iterativa es sencilla de implementar y mucho más rápida:

```
long long Fibonacci (int n){  
    long long anterior = 1, pre_anterior = 1;  
    long long actual;  
  
    if (n == 0 || n == 1)  
        actual = 1;  
    else  
        for (int i = 2; i <= n; i++) {  
            actual = anterior + pre_anterior;  
            pre_anterior = anterior;  
            anterior = actual;  
        }  
  
    return actual;  
}
```

Ampliación:

Una forma alternativa de evitar el problema de los cómputos repetidos en recursividad, lo proporcionan las técnicas de la *programación dinámica (dynamic programming)* que, a grosso modo, van guardando adecuadamente los resultados obtenidos previamente.



Caso base demasiado pequeño

Muchos problemas recursivos tienen como caso base un problema de un tamaño reducido. En ocasiones es *excesivamente* pequeño.

Para resolverlo, se suele aplicar un algoritmo iterativo cuando se ha llegado a un caso base suficientemente pequeño.

Ejemplo. Búsqueda binaria.

Aplicamos el algoritmo recursivo en general y el iterativo cuando el tamaño del subvector en el que estoy buscando es pequeño.

Recorrer las componentes con dos apuntadores izda y dcha

Si el número de elementos entre izda y dcha es pequeño

Buscar el valor con un algoritmo no recursivo

en otro caso

Colocarse en el centro y comprobar si es el valor buscado

Si no lo es

Buscar a la izda o a la derecha de centro

(realizando la llamada recursiva)

según sea el elemento buscado menor o mayor que v[centro]

```
class MiVectorEnteros{  
    .....  
    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;
}

int BusquedaBinaria(int izda, int dcha, int buscado){
    const int umbral = 5;
    int centro;

    if (dcha - izda < umbral)
        return PrimeraOcurrenciaEntre(izda, dcha, buscado);
    else{
        centro = (izda + dcha) / 2;

        if (vector_privado[centro] == buscado)
            return centro;
        else
            if (vector_privado[centro] > buscado)
                return BusquedaBinaria(izda, centro-1, buscado);
            else
                return BusquedaBinaria(centro + 1, dcha, buscado);
    }
}
};
```

Nota. El caso base ($izda > dcha$) de la función ya no es necesario.

Ejemplo. (Ampliación) Mejoramos QuickSort de forma análoga:

```
class MiVectorEnteros{  
private:  
    int partir(int primero, int ultimo)  
    .....  
public:  
    .....  
    void Ordena_por_Seleccion_entre(int primero, int ultimo){  
        int pos_min;  
  
        for (int izda=primero ; izda<ultimo ; izda++){  
            pos_min = PosicionMinimoEntre (izda, ultimo);  
            IntercambiaComponentes_en_Posiciones (izda, pos_min);  
        }  
    }  
    void QuickSort(int inicio, int final){  
        const int umbral = 5;  
        int pos_pivote;  
  
        if (final - inicio < umbral)  
            Ordena_por_Seleccion_entre(inicio, final);  
        else{  
            if (inicio < final) {  
                pos_pivote = partir (inicio, final);  
                QuickSort (inicio, pos_pivote - 1);      // Ordena primera mitad  
                QuickSort (pos_pivote + 1, final);        // Ordena segunda mitad  
            }  
        }  
    }  
};
```

Nota. Para Quicksort, lo usual es usar umbrales entre 4 y 15.

Problemas complejos con una solución recursiva sencilla

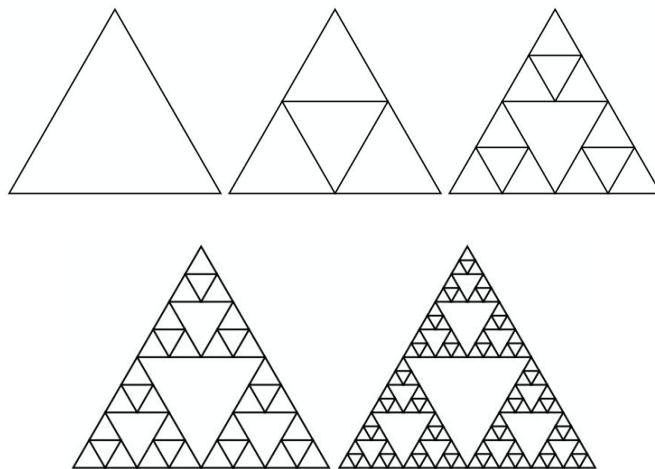
Algunos problemas se resuelven directamente con recursividad. Normalmente, también es posible resolverlos iterativamente, pero puede resultar más complejo.

Ejemplo. Definición de *fractal (fractal)* de la Wikipedia:

Un fractal es un objeto semigeométrico cuya estructura básica, fragmentada o irregular, se repite a diferentes escalas. A un objeto geométrico fractal se le atribuyen las siguientes características:

- ▷ **Es demasiado irregular para ser descrito en términos geométricos tradicionales.**
- ▷ **Es autosimilar.- Su forma es hecha de copias más pequeñas de la misma figura. Las copias son similares al todo: misma forma pero diferente tamaño.**
- ▷ ...
- ▷ **Se define mediante un simple algoritmo recursivo.**

Ejemplo. Triángulos de Sierpinski



```
class Sierpinski{
public:
    void Triangulo(Punto2D A, Punto2D B, Punto2D C, int grado){
        PintorFiguras pintor;

        if (grado == 0)
            pintor.DibujaTriangulo (A, B, C);
        else{
            SegmentoDirigido segmentoAB(A, B);
            SegmentoDirigido segmentoBC(B, C);
            SegmentoDirigido segmentoCA(C, A);

            Punto2D MedioAB (segmentoAB.Medio());
            Punto2D MedioBC (segmentoBC.Medio());
            Punto2D MedioCA (segmentoCA.Medio());

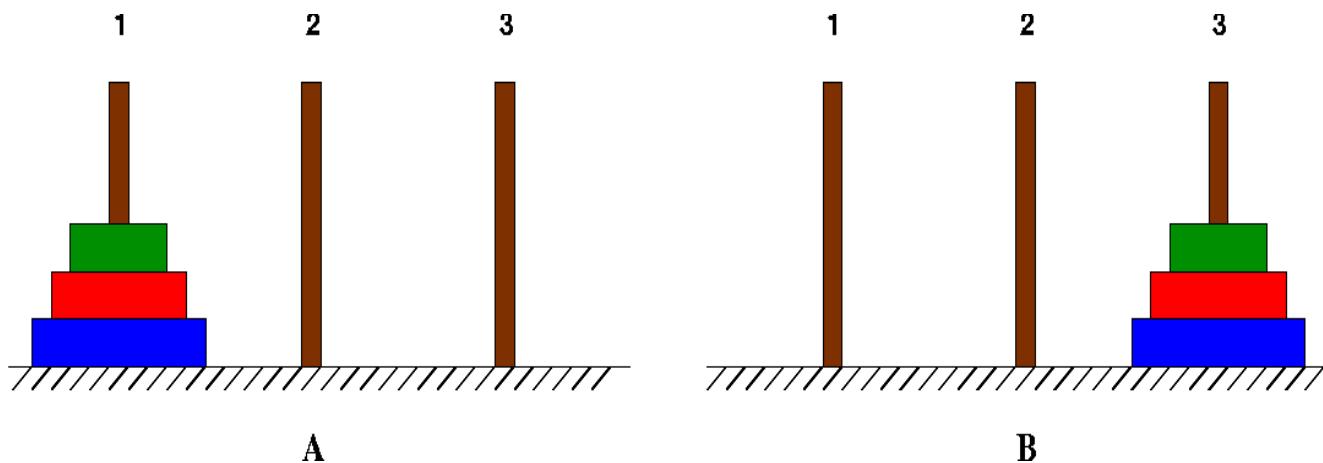
            Triangulo (A, MedioAB, MedioCA, grado - 1);
            Triangulo (MedioAB, B, MedioBC, grado - 1);
            Triangulo (MedioCA, MedioBC, C, grado - 1);
        }
    }
};
```

Ejemplo. Torres de Hanoi. Edouard Lucas, 1883.



Se trata de mover un conjunto de discos apilados en un poste a otro poste, con las condiciones siguientes:

- ▷ Se tienen 3 postes (hay versiones para trabajar con más postes)
- ▷ Sólo se puede mover un disco en cada paso. Además, debe ser el disco que está en la parte superior de la pila.
- ▷ Los discos en cualquier poste siempre deben formar una pirámide, es decir, colocados de mayor a menor.



Etiquetando los 3 postes como inicial, intermedio, final, el algoritmo recursivo que lo resuelve sería:

Mover $n-1$ discos desde inicial hasta temporal usando final como intermedio
Mover el disco que queda en inicial hasta final
Mover $n-1$ discos desde intermedio hasta final usando inicial como intermedio

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

void Hanoi (int num_discos, int inicial, int intermedio, int final){
    if (num_discos > 0){
        Hanoi (num_discos-1, inicial, final, intermedio);
        cout << "\nDel poste " << inicial << " al " << final;
        Hanoi (num_discos-1, intermedio, inicial, final);
    }
}
int main(){
    int n; // Número de discos a mover

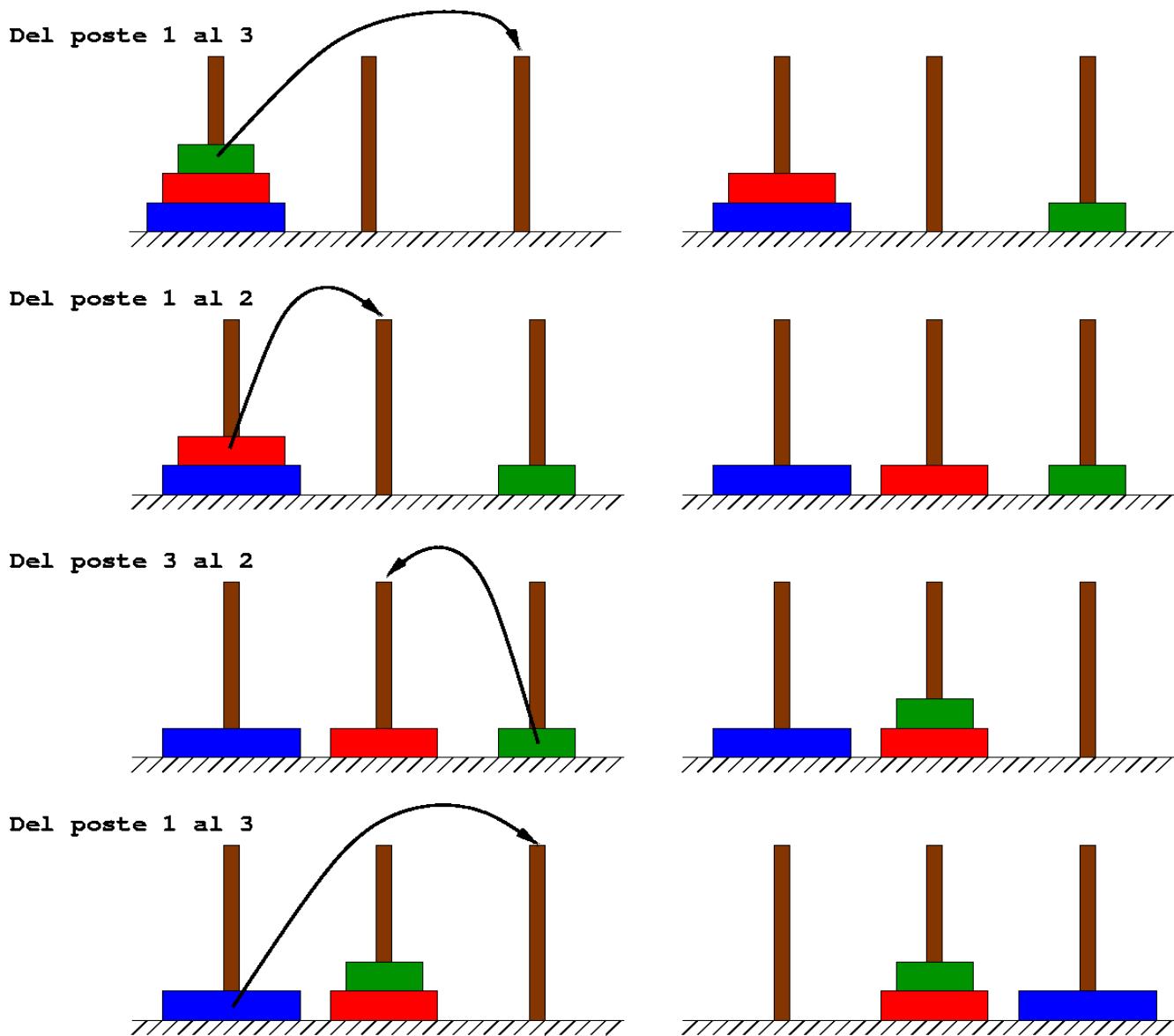
    cout << "Número de discos: ";
    cin >> n;

    Hanoi (n, 1, 2, 3); // mover "n" discos del 1 al 3,
                        // usando el 2 como temporal.
}
```

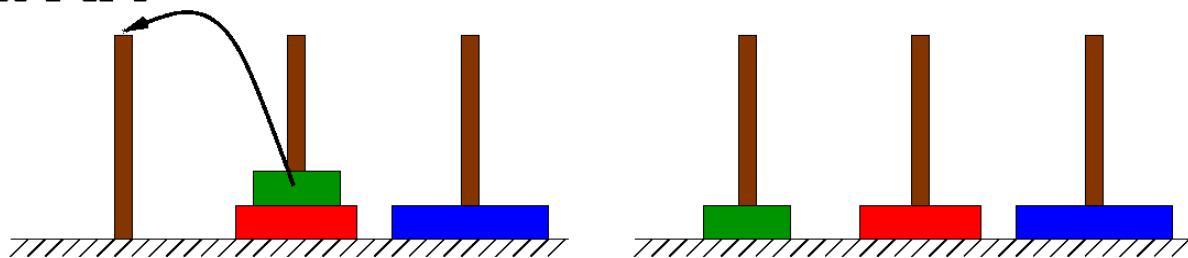
Su ejecución para tres discos produce el siguiente resultado:

```
Numero de discos: 3
Del poste 1 al 3
Del poste 1 al 2
Del poste 3 al 2
Del poste 1 al 3
Del poste 2 al 1
Del poste 2 al 3
Del poste 1 al 3
```

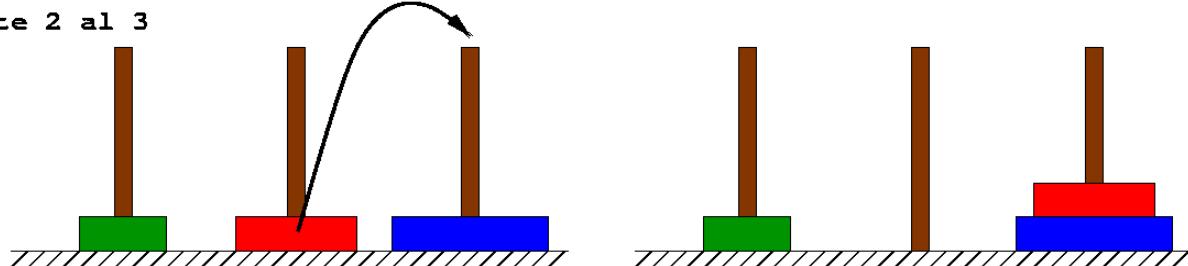
En definitiva, bastan siete movimientos para resolver el problema. Gráficamente se entenderá mejor:



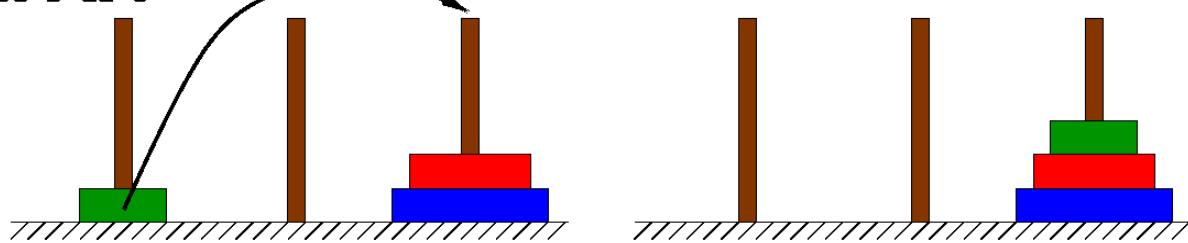
Del poste 2 al 1



Del poste 2 al 3



Del poste 1 al 3



Algunas curiosidades:

- ▷ **Applet:** <http://www.mazeworks.com/hanoi/index.htm>
- ▷ **Número de movimientos:** 2^k , siendo k = número de discos. Para 64 discos, moviendo 1 disco por segundo, se tardaría algo menos de 600 mil millones de años (el Universo tiene unos 20 mil millones)
- ▷ También existe una versión iterativa. No son tres líneas, pero no es demasiado complicada. En cualquier caso, el número de movimientos es el mismo.

