

# Prácticas con C++

## Metodología de la Programación

Antonio Garrido Carrillo

**eug**  
EDITORIAL  
UNIVERSIDAD  
DE GRANADA



© ANTONIO GARRIDO CARRILLO  
© UNIVERSIDAD DE GRANADA  
PRÁCTICAS CON C++: Metodología de la Programación  
ISBN: 978-84-338-5884-9.  
Edita: Editorial Universidad de Granada.  
Campus Universitario de Cartuja. Granada.

*Printed in Spain*

*Impreso en España.*

Cualquier forma de reproducción, distribución, comunicación pública o transformación de esta obra sólo puede ser realizada con la autorización de sus titulares, salvo excepción prevista por la ley.

# Índice general



<b>1</b>	<b>Tipos aritméticos</b>	<b>1</b>
1.1	Objetivos	1
1.2	Rango y precisión	1
1.2.1	Tamaños	1
1.2.2	Límites numéricos	2
1.3	Aritmética en punto flotante	2
1.3.1	Valores especiales	3
1.4	E/S formateada	3
1.5	Conversiones	4
1.5.1	El tipo booleano	4
1.5.2	El tipo carácter	5
1.5.3	Conversiones explícitas	5
1.6	Experimentando con la codificación	6
1.6.1	Codificación de caracteres	6
1.6.2	Codificación de enteros y reales: uniones	7
<b>2</b>	<b>Gestión de proyectos</b>	<b>9</b>
2.1	Introducción	9
2.1.1	Problema: círculo central	9
2.1.2	Compilación y ejecución	10
2.2	Módulos y compilación separada	10
2.2.1	Separación en archivos	11
2.2.2	Compilación, enlazado y ejecución	12
2.2.3	Bibliotecas	13
2.3	Gestión del proyecto con make	15
2.3.1	Dependencias entre módulos	15
2.3.2	Archivo <i>makefile</i>	16
2.3.3	Reglas implícitas	21
2.4	Mejora y ampliación de la biblioteca	22
2.4.1	Programas a desarrollar	22
2.4.2	Práctica a entregar	26

<b>3</b>	<b>Esteganografía: texto</b>	<b>27</b>
3.1	<b>Introducción</b>	<b>27</b>
3.1.1	Tipo enumerado . . . . .	27
3.1.2	Operadores a nivel de bit . . . . .	28
<b>3.2</b>	<b>Imágenes</b>	<b>29</b>
3.2.1	Niveles de gris y color . . . . .	30
3.2.2	Funciones de E/S de imágenes . . . . .	31
<b>3.3</b>	<b>Ocultar/Revelar un mensaje</b>	<b>32</b>
<b>3.4</b>	<b>Desarrollo de la práctica</b>	<b>32</b>
3.4.1	Módulo codificar . . . . .	33
3.4.2	Programas . . . . .	33
3.4.3	Práctica a entregar . . . . .	34
<b>4</b>	<b>Matriz de booleanos</b>	<b>35</b>
<b>4.1</b>	<b>Introducción</b>	<b>35</b>
4.1.1	Condiciones de desarrollo . . . . .	35
<b>4.2</b>	<b>Problema a resolver</b>	<b>35</b>
4.2.1	Ejemplos de ejecución . . . . .	36
4.2.2	Formato de archivos . . . . .	37
<b>4.3</b>	<b>Diseño propuesto</b>	<b>37</b>
4.3.1	Módulo <i>MatrizBit</i> . . . . .	38
4.3.2	Comprobando la abstracción . . . . .	41
<b>4.4</b>	<b>Práctica a entregar</b>	<b>42</b>
<b>5</b>	<b>Algoritmos con vectores</b>	<b>45</b>
<b>5.1</b>	<b>Introducción</b>	<b>45</b>
5.1.1	Objetivos . . . . .	45
5.1.2	Condiciones de desarrollo . . . . .	45
<b>5.2</b>	<b>Vectores en memoria dinámica</b>	<b>46</b>
5.2.1	Vector dinámico . . . . .	46
5.2.2	Búsqueda . . . . .	50
5.2.3	Otros algoritmos de ordenación . . . . .	54
5.2.4	Rangos de elementos . . . . .	58
<b>6</b>	<b>Celdas enlazadas</b>	<b>63</b>
<b>6.1</b>	<b>Introducción</b>	<b>63</b>
6.1.1	Objetivos . . . . .	63
6.1.2	Condiciones de desarrollo . . . . .	63
<b>6.2</b>	<b>Lista de celdas enlazadas</b>	<b>64</b>
6.2.1	Búsqueda, inserción y borrado . . . . .	65
6.2.2	Celda controlada desde la anterior . . . . .	67
6.2.3	Ordenación . . . . .	68
6.2.4	Rangos de elementos . . . . .	71
<b>7</b>	<b>Conecta N</b>	<b>73</b>
<b>7.1</b>	<b>Introducción</b>	<b>73</b>
7.1.1	El juego <i>Conecta-4</i> . . . . .	73
<b>7.2</b>	<b>Problema a resolver</b>	<b>74</b>
7.2.1	Puntuación de las partidas . . . . .	74
7.2.2	Ejemplo de ejecución . . . . .	74

<b>7.3</b>	<b>Diseño propuesto: versión 1</b>	<b>77</b>
7.3.1	Interfaces e implementación . . . . .	77
7.3.2	Programa de la versión 1 . . . . .	79
<b>7.4</b>	<b>Modificación del programa: versión 2</b>	<b>79</b>
7.4.1	Cambios internos a un módulo . . . . .	79
7.4.2	Cambios en la interfaz de un módulo . . . . .	80
7.4.3	Ampliar la funcionalidad del programa . . . . .	81
<b>7.5</b>	<b>Práctica a entregar</b>	<b>84</b>
<b>A</b>	<b>Guía de Estilo</b> . . . . .	<b>87</b>
<b>A.1</b>	<b>Introducción</b>	<b>87</b>
<b>A.2</b>	<b>Indicaciones generales</b>	<b>87</b>
A.2.1	Énfasis automático . . . . .	88
A.2.2	Reglas de estilo . . . . .	88
<b>A.3</b>	<b>Guía de estilo</b>	<b>88</b>
A.3.1	Identificadores . . . . .	89
A.3.2	Estructura del código . . . . .	92
A.3.3	Consideraciones adicionales . . . . .	97
<b>B</b>	<b>Generación de números aleatorios</b> . . . . .	<b>103</b>
<b>B.1</b>	<b>Introducción</b>	<b>103</b>
<b>B.2</b>	<b>El problema</b>	<b>103</b>
B.2.1	Números pseudoaleatorios . . . . .	104
<b>B.3</b>	<b>Transformación del intervalo</b>	<b>105</b>
B.3.1	Operación módulo . . . . .	105
B.3.2	Normalizar a U(0,1) . . . . .	105
<b>C</b>	<b>Tablas</b> . . . . .	<b>107</b>
<b>C.1</b>	<b>Tabla ASCII</b>	<b>107</b>
<b>C.2</b>	<b>Operadores C++</b>	<b>108</b>
<b>C.3</b>	<b>Palabras reservadas de C89, C99, C11, C++ y C++11</b>	<b>109</b>
<b>C.4</b>	<b>Manipuladores y funciones miembro para E/S formateada</b>	<b>109</b>
C.4.1	Banderas y máscaras . . . . .	110
C.4.2	Funciones miembro y manipuladores . . . . .	111
	<b>Bibliografía</b> . . . . .	<b>113</b>
	<b>Referencias electrónicas</b>	<b>113</b>
	<b>Índice alfabético</b> . . . . .	<b>115</b>



# Prólogo



El objetivo de este cuaderno es ofrecer un guión para realizar las prácticas relacionadas con el libro *Metodología de la Programación*, que se usa en los estudios de la *Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación* de la *Universidad de Granada*.

Los nuevos planes de estudios —donde se enfatiza el trabajo autónomo del alumno— junto con las necesidades de asignaturas con una carga práctica tan importante, hace necesario crear documentos suficientemente detallados que sirvan como guía autocontenido de los ejercicios y proyectos que el estudiante debe realizar.

## Organización del cuaderno de prácticas

La secuencia de capítulos se ha organizado para cubrir los contenidos que se estudian en el libro. Cada capítulo corresponde a un guión de prácticas, comenzando por un guión básico sobre los tipos más simples que ofrece el lenguaje hasta llegar a un proyecto suficientemente complejo como para mostrar las ideas básicas de la metodología de la programación a un alumno de primer curso. Más concretamente, se llevan a cabo prácticas que incorporan contenidos sobre:

- Gestión de proyectos en C++.
- Tipos de datos simples, incluyendo algoritmos que trabajan a nivel de bits.
- Vectores-C y cadenas-C.
- Estructuras.
- Matrices-C.
- Memoria dinámica.
- Conceptos de abstracción.
- Encapsulación en clases simples.
- Desarrollo avanzado de clases, incluyendo memoria dinámica.

Los contenidos teóricos para desarrollar las prácticas se ofrecerán en las clases de teoría o en el laboratorio cuando se desarrolle cada práctica. En general, el cuaderno no incluye contenidos sobre el lenguaje, excepto algunos casos excepcionales en los que se complementa la exposición del problema con alguna forma de resolverlo en C++, ya sea para puntualizar algún aspecto, recordar brevemente alguna idea o para facilitar el uso de alguna utilidad o biblioteca.

Lo que se incluye es una exposición bastante detallada del problema para que el alumno pueda consultarla y revisarla de forma autónoma. Lógicamente, un alumno de primer curso puede encontrarse con muchas dudas, que deberán ser resueltas por el tutor o profesor. Probablemente, se encontrará con el habitual “*¿Por dónde empiezo?*”, especialmente si tiene poca experiencia. Para facilitar estos casos, los guiones contendrán indicaciones sobre el diseño que se solicita, indicando a un nivel bastante detallado qué tipo de solución se desea.

Finalmente, el documento incluye apéndices con información relevante para realizar los guiones. En concreto, podrá usarlos para consultar:

- Una *guía de estilo*. Es importante escribir código con un estilo claro y eficaz. Desde las primeras clases de programación, ya aparecen distintas alternativas a la hora de formatear nuestro código. En este curso, más avanzado, debería establecerse un criterio común que garantice que el código sea legible y que se pueda compartir con otros programadores. Es este apéndice se comentan algunas alternativas y se ofrecen algunos consejos para homogeneizar el código que se presenta en este documento y que se recomienda a los estudiantes.
- La *generación de números aleatorios*. Es un tema que generalmente no se aborda directamente en las clases de teoría, sino que se supone se practicará cuando se desarrollen programas que generan valores aleatorios. Sin embargo, es un

tema cuyo contenido teórico es muy relevante para poder usarlo adecuadamente. En lugar de dar una breve especificación de las funciones que ofrece el lenguaje, se incluye una exposición más detallada con el fin de que el estudiante no sólo lo use, sino que entienda por qué funciona.

- *Tablas* relacionadas con el lenguaje. En la práctica, es muy útil disponer de tablas que incluyen detalles sobre palabras reservadas, operadores del lenguaje, código *ASCII*, etc.

## Agradecimientos

Este prólogo no puede terminar sin el agradecimiento a los alumnos. Por un lado, a los que con su interés por aprender constituyen una motivación para dedicar el tiempo y esfuerzo necesario para realizar este tipo de trabajos. Por otro lado, a los que con su paciencia y esfuerzo han sabido agradecer el trabajo realizado para facilitar su aprendizaje. Este agradecimiento es especialmente sentido cuando la recompensa por todo este trabajo viene casi exclusivamente de ellos... sin el “casi”.

A. Garrido.  
Febrero de 2016.

Objetivos .....	1
Rango y precisión .....	1
Tamaños	
Límites numéricos	
Aritmética en punto flotante .....	2
Valores especiales	
E/S formateada .....	3
Conversiones .....	4
El tipo booleano	
El tipo carácter	
Conversiones explícitas	
Experimentando con la codificación .....	6
Codificación de caracteres	
Codificación de enteros y reales: uniones	

## 1.1 Objetivos

El objetivo de esta práctica es que el alumno conozca con más precisión las características de los tipos aritméticos en C++. Recordemos que estos tipos se clasifican en:

1. Tipos integrales: booleanos, carácter y enteros.
2. Tipos en coma flotante.

En esta práctica profundizaremos en las características de estos tipos de datos, completando los conocimientos que se han visto en clase de teoría. El alumno deberá revisarlo y comprender los distintos conceptos y detalles que se muestran. De esta forma, podrá entender mejor el comportamiento de sus programas. Después de realizar esta práctica, el alumno deberá haber asimilado:

1. Los distintos tipos y sus relaciones, teniendo en cuenta sus distintos tamaños y precisiones. Aunque no será habitual usar `sizeof` y `numeric_limits`, el alumno debe saber que existen métodos que nos permiten consultar las características de los distintos tipos.
2. La aritmética en punto flotante implica una aproximación de la recta real y, por tanto, el cálculo con valores aproximados.
3. Existen los valores especiales para un número no representable (`Nan`) e infinito (`Inf`) para los tipos en punto flotante. Aunque no los usaremos explícitamente, el alumno debe conocer que pueden aparecer en sus programas y entender a qué se refieren.
4. El lenguaje ofrece un grupo de herramientas que permiten el formateo de la salida.
5. Es importante conocer en qué consisten las conversiones. De especial interés son las que involucran a booleanos y caracteres, pues son tipos que, en principio, no parecen relacionados con el resto de aritméticos.
6. Se puede usar la conversión explícita para que el compilador haga, exactamente, lo que el programador desea.

## 1.2 Rango y precisión

El rango y precisión de un tipo depende de la plataforma en la que se está desarrollando. El estándar C++ no garantiza más que los mínimos siguientes:

Además, garantiza que esos tamaños están “ordenados”. Por ejemplo, el tamaño de un tipo `float` no puede ser mayor que el de un tipo `double`. Se podrían listar todos los tipos de datos: `signed char, char, unsigned char, signed short int, signed int, signed long int, unsigned short int, unsigned int, unsigned long int, float, double, long double`, incluyendo modificadores de signo, aunque un tipo con signo o sin él ocupa lo mismo. Observe que en el caso de los enteros, el no incluir modificador equivale a añadir `signed`.

### 1.2.1 Tamaños

El operador `sizeof` del lenguaje C++ nos permite obtener el tamaño de un determinado tipo u objeto. En un programa podemos escribir este operador con el nombre de un tipo entre paréntesis para obtener el tamaño<sup>1</sup> de dicho tipo.

<sup>1</sup>El tamaño corresponde al número de “char” que ocupa. Es decir, el tamaño de `char` es 1 y el del resto un múltiplo de lo que ocupa éste. En nuestro caso, en el que `char` ocupa un byte, podemos pensar en número de bytes.

**Tabla 1.1**  
Tamaño mínimo de los tipos básicos.

Tipo	Tamaño mínimo
char	1
short int	2
int	2
long int	4
float	4
double	8
long double	8
long long int (desde C++11)	8

**Ejercicio 1.1 — Operador sizeof.** Escriba un programa que imprima, ordenadamente, los tamaños de los distintos tipos para la plataforma en la que está trabajando.

Puede comprobar —con una simple modificación— que si a los tipos que ha escrito le antepone alguna modificación de signo, `signed` o `unsigned`, obtiene exactamente los mismos tamaños.

### 1.2.2 Límites numéricos

Si queremos que nuestro programa tenga información concreta sobre los límites exactos que presentan los distintos tipos, podemos usar la biblioteca estándar de C++.

En primer lugar, debemos tener en cuenta que en C++ también tenemos disponibles las mismas constantes que se usan en C. Así, los archivos de cabecera `limits.h` y `float.h` están disponibles en C++ como `climits` y `cfloat`.

Sin embargo, en C++ vamos a preferir el uso de la plantilla `numeric_limits` para obtener esta información, previa inclusión del archivo de cabecera `limits`. Aunque entender cómo funciona esta plantilla es un tema avanzado, podemos usarla para implementar un programa simple que nos informe de algunos valores usados en nuestro sistema. Para ello, tenga en cuenta que la forma en que podemos usarla tiene el formato:

`numeric_limits<TIPO>::MIEMBRO`

donde `TIPO` es el nombre del tipo que queremos consultar y `MIEMBRO` se refiere a la información que queremos obtener. Algunos miembros que podríamos consultar son:

1. Para tipos entero y en coma flotante los valores del mínimo y máximo del rango: `min()`, `max()`.
2. Para tipos en coma flotante la diferencia entre uno y el menor valor representable mayor que uno: `epsilon()`.

**Ejercicio 1.2 — Límites numéricos.** Escriba un programa que imprima los valores de los miembros que se han listado para los distintos tipos: `signed short int`, `signed int`, `signed long int`, `unsigned short int`, `unsigned int`, `unsigned long int`, `float`, `double`, `long double`.

**Ejercicio 1.3 — Rango limitado de enteros.** Escriba un programa que declare un objeto de tipo entero, le asigne el valor máximo, le sume uno, y finalmente lo imprima. A continuación, le asigne el mínimo, le reste uno, y lo imprima. ¿Qué resultados obtiene?

## 1.3 Aritmética en punto flotante

Analice el siguiente programa:

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

int main()
{
    char son_iguales;
    double x;
    cout << "Dame un número y comprobaré si se cumple "
        "que la raíz cuadrada al cuadrado da el número: ";
    cin >> x;
    son_iguales = ( sqrt(x)*sqrt(x) == x ) ? 'S' : 'N';
    cout << "Son iguales: " << son_iguales << endl;
}
```

Para ello, compruebe el resultado de su ejecución para distintos números. Por ejemplo, puede comprobarlo para el número 2 y el 4.

**Ejercicio 1.4 — Precisión de números en coma flotante.** En el programa anterior es posible obtener valores que no son iguales. ¿Por qué? Modifique el programa anterior para que funcione correctamente en todos los casos.

### 1.3.1 Valores especiales

Los números en coma flotante se almacenan habitualmente siguiendo el estándar IEEE-754. Dicha representación permite almacenar algunos valores especiales como:

- **Nan** (Not a Number). Se usa cuando hay que almacenar algo que no se corresponde con un número real válido.
- **Inf**. Representa el valor  $\infty$  (infinito).

Podemos usar **numeric\_limits** para trabajar con estos valores especiales. En particular, pueden ser interesantes las siguientes propiedades:

- **quiet\_NaN()**. Devuelve el valor **Nan**.
- **infinity()**. Devuelve el valor **Inf**.
- **is\_iec559**. Devuelve true si el compilador está usando el estándar IEEE-754/IEC-559. Si lo cumple, entonces tenemos garantizado que la comparación entre un **Nan** y cualquier otra cosa (incluido el propio **Nan**) es false.

El siguiente programa ilustra estos aspectos:

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

int main()
{
    double n1 = numeric_limits<double>::quiet_NaN();
    double n2 = numeric_limits<double>::infinity();
    double n3 = -numeric_limits<double>::infinity();
    double n4 = 2.7;

    cout << "n1 vale " << n1 << endl;
    cout << "n2 vale " << n2 << endl;
    cout << "n3 vale " << n3 << endl;
    cout << "n4 vale " << n4 << endl;

    cout << "Raíz de -1 vale " << sqrt(-1.0) << endl; // NaN
    cout << "0.0/0.0 vale " << 0.0/0.0 << endl; // NaN
    cout << "1e1000 vale " << 1e1000 << endl; // Inf
    cout << "-1e1000 vale " << -1e1000 << endl; // -Inf

    cout << "Representación según el estándar IEEE-754 / IEC-559 : " <<
        (numeric_limits<double>::is_iec559 ? "Si" : "No") << endl;

    cout << "n1 es NaN : " << (n1==n1 ? "No es NaN" : "Si es NaN") << endl;
    cout << "n4 es NaN : " << (n4==n4 ? "No es NaN" : "Si es NaN") << endl;

    cout << "n2 es Inf : " <<
        (n2==numeric_limits<double>::infinity() ? "Si es Inf" : "No es inf")
        << endl;
    cout << "n4 es Inf : " <<
        (n4==numeric_limits<double>::infinity() ? "Si es Inf" : "No es inf")
        << endl;
}
```

**Ejercicio 1.5 — Valores especiales de coma flotante.** Pruebe el programa anterior, revisando los valores que va obteniendo junto con las correspondientes líneas que los producen.

## 1.4 E/S formateada

En ocasiones podemos cambiar el formato o apariencia de lo que mostramos en consola mediante **cout**. En particular podemos hacer uso de lo que se conocen como “manipuladores de formato”. Se definen en el fichero de cabecera **iomanip**. Aquí presentamos algunos<sup>2</sup> a modo de ejemplo:

- **dec**. Permite mostrar números en base decimal (activo por defecto).
- **hex**. Permite mostrar números en base hexadecimal.
- **oct**. Permite mostrar números en base octal.
- **setw(x)**. Indica el ancho (número de caracteres) que debe ocupar en consola un determinado dato (sólo afecta al siguiente dato).
- **setfill(c)**. Indica el carácter que se usará para llenar el espacio que ocupa un dato (por defecto es un espacio).
- **left/right**. Permite justificar un dato a izquierda o derecha.

El siguiente programa ilustra el uso de algunos de estos manipuladores:

<sup>2</sup>Puede consultar la lista completa de manipuladores en <http://en.cppreference.com/w/cpp/io/manip>.

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

int main()
{
    int x;
    cout << "Dame un número: ";
    cin >> x;

    cout << "Decimal      : " << dec << x << endl;
    cout << "Octal       : " << oct << x << endl;
    cout << "Hexadecimal: " << hex << x << endl;

    cout << dec;

    cout << "8 posiciones      : " << setw(8) << x << endl;
    cout << "8 posiciones (just dcha): " << setw(8) << right << x << endl;
    cout << "8 posiciones (just izda): " << setw(8) << left << x << endl;

    double y;
    cout << "Dame otro número: ";
    cin >> y;

    cout << "Con 2 decimales: " << fixed << setprecision(2) << y << endl;
    cout << "Con 6 decimales: " << fixed << setprecision(6) << y << endl;
}
```

**Ejercicio 1.6 — Uso de manipuladores de formato.** A continuación tienes un programa que muestra dos listados de datos.

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Nombre" << "Apellidos" << "Edad" << "Estado" << endl;
    cout << "Javier" << "Moreno" << 20 << "S" << endl;
    cout << "Juan" << "Espejo" << 8 << "S" << endl;
    cout << "Antonio" << "Caballero" << 53 << "C" << endl;
    cout << "Jose" << "Cano" << 27 << "C" << endl;

    cout << endl;

    cout << 123.456 << 26.467872 << 876.3876 << endl;
    cout << 17.26734 << 0.22 << 18972.1 << endl;
    cout << 456.5 << 2897.0 << 2832.3 << endl;
}
```

Modifíquelo de manera que únicamente añadiendo manipuladores de formato, obtengamos una salida como la que se muestra a continuación:

Nombre	Apellidos	Edad	Estado
Javier	Moreno	20	S
Juan	Espejo	8	S
Antonio	Caballero	53	C
Jose	Cano	27	C
<hr/>			
123.46	26.47	876.39	
17.27	0.22	18972.10	
456.50	2897.00	2832.30	

En la sección C.4 (página 109) puede encontrar un listado más completo de funciones y manipuladores que puede usar para formatear la E/S. Además, se incluyen algunas indicaciones para que entienda la forma y sentido de esas órdenes.

## 1.5 Conversiones

Cuando en una expresión mezclamos distintos tipos de datos, el compilador se encarga de analizar la expresión y realizar las conversiones que sean necesarias. Estas conversiones se denominan implícitas; las realiza el compilador de forma automática.

### 1.5.1 El tipo booleano

El tipo **bool** es un tipo que no existe en C, aunque se incluye en lenguaje C++. En el primero, el manejo de valores de tipo booleano (**true/false**) se realiza mediante algún tipo integral, codificando el valor **false** como cero y el valor **true** como distinto de cero. Por ejemplo, si prescindimos del tipo **bool**, podríamos escribir:

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

```

int main()
{
    int es_menor;
    es_menor= 2<3;
    cout << "Resultado de 2<3: " << (es_menor ? "Si" : "No") << endl;
}

```

En C++ se pueden usar las mismas expresiones para representar un tipo booleano, por lo que debemos tener en cuenta esta relación para poder interpretar correctamente el comportamiento de nuestros programas.

**Ejercicio 1.7 — Conversión de bool y enteros.** Considere el siguiente programa:

```

#include <iostream>
using namespace std;
int main()
{
    cout << (4<1<5 ? "Ordenados" : "Desordenados") << endl;
}

```

Compruebe su comportamiento. ¿Qué está ocurriendo?

Aunque nosotros siempre usaremos el tipo **bool** para representar booleanos, debemos tener en cuenta que se pueden realizar conversiones implícitas entre los distintos tipos. Además, el compilador puede usar otros tipos de datos como valores booleanos, considerando que el valor cero corresponde a falso y distinto de cero a verdadero.

### 1.5.2 El tipo carácter

El tipo carácter en un tipo **integral** que se puede usar como un entero de menor tamaño. En principio, podemos considerar 3 tipos: **char**, **signed char** y **unsigned char**, aunque el primero de ellos se comportará como uno de los otros dos.

Es posible “mezclar” este tipo con un tipo entero de forma que el compilador puede convertir caracteres en enteros y enteros en caracteres. Tenga en cuenta que:

- Si en una expresión hay que operar un carácter con un entero, el compilador convierte el carácter a entero.
- Si asignamos un entero a un carácter, la asignación tendrá sentido si el valor del entero está en el rango válido del carácter.

De hecho, puede consultar las funciones que ofrece el fichero de cabecera **cctype**, donde puede comprobar que las funciones, aun presentándose como funciones que trabajan con caracteres, realmente trabajan con datos enteros. Este tipo de parámetros puede incluso pasar desapercibido, ya que el compilador realiza automáticamente las conversiones.

Aunque **char** sea un “entero pequeño”, **cout** distingue si lo que recibe es un carácter o un entero. Si tiene que imprimir el valor de una variable de tipo **int** sabe que debe imprimir los dígitos numéricos que corresponden a dicho valor, mientras que si recibe un **char**, sabe que debe imprimir el carácter correspondiente de la tabla **ASCII**<sup>3</sup>.

**Ejercicio 1.8 — Caracteres y número ASCII asociado.** Escriba un programa que recibe como entrada un número del 0 al 25. Como resultado deberá escribir la letra (el 0 indica la letra ‘**a**’ y el 25 la ‘**z**’), su correspondiente mayúscula y los dos valores **ASCII** correspondientes.

### 1.5.3 Conversiones explícitas

Hasta ahora siempre hemos considerado las conversiones como una tarea que realiza de forma automática el compilador. Sin embargo, es posible hacer una conversión explícita, es decir, indicar al compilador que queremos que convierta el valor de una expresión a un determinado tipo. En este caso, diremos que hacemos un *casting* (moldeado). La forma más simple de hacer un casting es con la siguiente sintaxis:

(TIPO) EXPRESIÓN

donde **TIPO** es el tipo al que queremos convertir la expresión. Por ejemplo, si deseamos escribir la parte entera de un valor real, podemos hacer:

```

#include <iostream>
using namespace std;

int main()
{
    double x;
    cout << "Dame un número: ";
    cin >> x;
    cout << "Parte entera: " << (int) x << endl;
}

```

<sup>3</sup>Véase la tabla extendida —codificación ISO-8859-15— en la figura C.1, página 107.

Tenga en cuenta que el *casting* no es más que otro operador que tiene una prioridad alta, ya que está al nivel de los operadores unarios (véanse los operadores en la tabla de la sección C.2, página 108). Si la expresión contiene otros operadores menos prioritarios, deberá ponerla entre paréntesis.

El operador de moldeado que acabamos de presentar es el que se usa en C y que también está disponible en C++. Sin embargo, en C++ se incorporan nuevos operadores que son más seguros<sup>4</sup>, ya que la conversión se diversifica con distintos tipos de *casting* de forma que el compilador conozca mejor nuestras intenciones. Aunque hay varios, simplemente comentaremos la sintaxis del único que usaremos en esta asignatura:

```
static_cast<TIPO>(EXPRESIÓN)
```

que convierte el valor resultante de la expresión al tipo **TIPO**.

**Ejercicio 1.9 — Caracteres y número ASCII asociado.** Escriba un programa que recibe como entrada un número del 0 al 25. Como resultado deberá escribir la letra (el 0 indica la letra '*a*' y el 25 la '*z*'), su correspondiente mayúscula y los dos valores *ASCII* correspondientes, habiendo declarado únicamente un objeto de tipo **int**.

**Ejercicio 1.10 — Límites del tipo char.** Escriba un programa que imprima los límites (mínimo y máximo) de los tres tipos de datos: **signed char**, **char** y **unsigned char**. Con este ejercicio podrá confirmar a qué tipo de dato corresponde el tipo **char** de su sistema.

## 1.6 Experimentando con la codificación

La mejor forma de entender que los datos se pueden codificar de distinta forma es con la práctica. Es conveniente que hagamos algunas pruebas para confirmar cómo se comporta nuestro sistema con algunos tipos de datos.

### 1.6.1 Codificación de caracteres

Por simplicidad, vamos a trabajar fundamentalmente considerando una codificación *ISO-8859-15*, es decir, con la tabla *ASCII extendida para Europa Occidental*. Por tanto, cuando trabajemos con texto, la lectura de un objeto de tipo **char** implicará la lectura de un byte que corresponde a un carácter de la tabla que se muestra en la figura C.1, página 107.

Sin embargo, en la práctica podemos editar en otros formatos. En concreto, la codificación *UTF8* es la más habitual en *GNU/Linux*. En las secciones anteriores se ha trabajado especialmente con caracteres que corresponden a la primera parte de la tabla *ASCII*. Estos caracteres se codifican de forma idéntica en ambos casos, por lo que hemos evitado considerar ningún detalle relativo a la codificación. Ahora bien, ¿Qué ocurre con otros caracteres de la segunda mitad de la tabla?

Los caracteres de la segunda mitad son un valor de 128 a 255 en la codificación *ISO-8859-15*, por lo que sólo necesitan de un byte. En código *UTF8*, se codifican como dos bytes.

**Ejercicio 1.11 — Analizando un archivo.** Escriba un programa **ver\_bytes.cpp** que lea desde la entrada estándar —desde **cin**— el contenido de un archivo y escriba en la salida estándar todos y cada uno de los valores de **char** que lea. El valor escrito será un número en el rango 0-255. Para probar el programa, obtenga los valores que genera un archivo **letras8859.txt** que contenga las siguientes letras (en formato *ISO-8859*):

a e i o u A E I O U á é í ó ú Á É Í Ó Ú ñ Ñ ü Ü

Los valores obtenidos deberían coincidir con la información que presenta la tabla de la figura C.1.

Realmente, el programa que acaba de crear en el ejercicio anterior para analizar valores de la tabla *ASCII* también es válido para analizar un archivo en *UTF8*. Puede probar el resultado sobre un archivo **letrasUTF8.txt** de idéntico contenido<sup>5</sup>, pero con esta codificación. Podrá comprobar que los caracteres que tenían códigos de la parte extendida ahora se codifican de otra forma.

Si analiza con cuidado los resultados de los valores numéricos que ha obtenido para ambas codificaciones, podrá entender que si tenemos un texto escrito en castellano con una de las dos codificaciones, probablemente se puede deducir a cuál de ellas corresponde. Esta operación la realizan muchos editores cuando abren un archivo, de forma que el usuario pueda trabajar con la misma codificación que contenía el archivo sin necesidad de saber nada sobre ello.

<sup>4</sup>No entraremos en detalles sobre los distintos tipos de *casting*, ya que corresponden a un tema avanzado.

<sup>5</sup>Puede usar la orden **iconv** de *GNU/Linux* para transformar cualquier fichero de una codificación a otra. Pruebe por ejemplo **iconv -l** y podrá ver la cantidad de codificaciones que podría procesar.

**Ejercicio 1.12 — Adivinando la codificación.** Escriba un programa `adivina_codigo.cpp` que lea desde la entrada estándar —desde `cin`— el contenido de un archivo y escriba en la salida estándar la posible codificación. Para simplificar el problema, suponga que sólo analizamos las letras de un texto escrito en castellano.

Recuerde que los códigos de las letras:

a e i o u A E I O U á é í ó ú Á É Í Ó Ú ñ Ñ ü Ü

son los siguientes:

- En codificación *ISO-8859-15*: 97, 101, 105, 111, 117, 65, 69, 73, 79, 85, 225, 233, 237, 243, 250, 193, 201, 205, 211, 218, 241, 209, 252 y 220.
- En codificación *UTF8*: 97, 101, 105, 111, 117, 65, 69, 73, 79, 85, (195/161), (195/169), (195/173), (195/179), (195/186), (195/129), (195/137), (195/141), (195/147), (195/154), (195/177), (195/145), (195/188) y (195/156).

Pruebe el resultado con los archivos de ejemplo `texto1.txt`, `texto2.txt`, `texto3.txt` y `texto4.txt`.

Observe que si nos limitamos a los caracteres habituales en un texto en castellano no hay demasiadas diferencias entre ambas codificaciones.

**Ejercicio 1.13 — Conversor.** Escriba un programa `utf2iso8859.cpp` que lea desde la entrada estándar —desde `cin`— el contenido de un archivo codificado en *UTF8* y escriba en la salida estándar el mismo archivo pero codificado en formato *ISO-8859-15*. Para probar el programa, use el archivo `textoUTF8.txt`.

## 1.6.2 Codificación de enteros y reales: uniones

Para experimentar con tipos de dato entero y real, vamos a introducir las *uniones*, una forma especial de estructura. Para definir una union podemos usar la siguiente sintaxis:

```
union NOMBRE_UNION {
    tipo1 campo1;
    tipo2 campo2;
    ...
    tipon campon;
};
```

Esta sintaxis es muy parecida a la de las estructuras. Define un nuevo tipo con el nombre de la `union`. Después de su definición, podemos declarar objetos con ese nuevo nombre. Por ejemplo, en el siguiente código:

```
union Mezcla {
    double x;
    int i;
};
Mezcla m;
```

se define un nuevo tipo, de nombre `Mezcla`, con dos miembros —de tipo `double` y tipo `int`— y a continuación declaramos el objeto `m` de ese tipo.

Las uniones están diseñadas para reservar un espacio de memoria tan grande como el más grande de los miembros que la componen. Cuando definimos una unión, no queremos almacenar tantos objetos como miembros hemos incluido en la definición. El objetivo es almacenar tan sólo uno de ellos. Por eso, el compilador sólo necesita reservar espacio suficiente para el mayor. Por ejemplo, en mi máquina el tipo `double` ocupa 8 bytes y el tipo `int` ocupa 4. El objeto `m` de tipo `Mezcla` anterior ocupa 8 bytes, es decir, el mayor de los tamaños.

El efecto de declarar `m` de tipo `Mezcla` es que el compilador busca una zona de memoria para la unión y la llama `m`. Una vez tenemos el objeto, podemos usar cualquiera de sus miembros para usar la zona de memoria como más nos convenga. Es decir, podemos decir que:

- Todos los miembros se sitúan en la misma dirección de memoria.
- Cada miembro o campo se refiere a una interpretación de los contenidos que hay en esa zona de memoria.
- En un momento dado, sólo nos interesa uno de los miembros de la estructura. Si almacenamos un dato conforme a uno de sus miembros, sólo tendría sentido usar el contenido de la unión con esa misma interpretación. Si se accede a otro de los miembros, el resultado está indeterminado.

Las uniones serán útiles cuando queramos almacenar sólo un dato, aunque dependiendo de alguna condición, podría corresponder a uno de entre varios tipos. Por ejemplo, en la unión que hemos definido podemos almacenar un entero o un número real, dependiendo del campo al que nos refiramos. Si almacenamos un `int`, no tiene sentido que preguntemos por el valor del campo `double`, ya que la única forma de garantizar que el dato tiene sentido es acceder a la unión como un entero.

Las uniones son poco habituales, especialmente en código a un nivel de abstracción alto. Muchas soluciones en base a uniones se pueden resolver de una forma más eficaz con otras técnicas. Es posible que si encuentra código de este tipo, sea a un nivel bajo, en casos en los que se desea resolver un problema de una forma muy simple, con un mínimo de memoria.

En este guión proponemos su uso para explorar los contenidos de distintos tipos. La idea es que si almacenamos en la unión un dato por uno de sus campos, podemos acceder a otro de sus campos para ver los efectos que tendría ese contenido interpretado de otra forma.

**Ejercicio 1.14 — Interpretación de la memoria.** Escriba un programa `interpretaciones.cpp` para analizar algunos detalles de representación de su máquina. Antes de resolver el problema:

1. Determine qué tipo de dato de su máquina es un entero sin signo de 1 byte.
2. Determine qué tipo de dato de su máquina es un entero sin signo de 2 bytes.
3. Determine qué tipo de dato de su máquina es un entero con signo de 4 bytes.
4. Determine qué tipo de dato de su máquina es un número en coma flotante con 4 bytes.

Nos centraremos en analizar los efectos en un bloque de 4 bytes. Este bloque puede interpretarse de distintas formas:

- Como un entero con signo.
- Como un número real de 4 bytes.
- Como 2 números enteros sin signo de 16 bits, consecutivos.
- Como 4 números enteros sin signo de 8 bits, consecutivos.

El programa debe definir un tipo de dato que permita consultar un objeto de 4 bytes con cualquiera de las 4 interpretaciones. Para ello, defina una unión que tenga las 4 posibilidades. Use los tipos de datos y definiciones que considere necesarios. El programa debe:

1. Imprimir en la salida estándar el tamaño de ese tipo de dato para confirmar que tiene 4 bytes.
2. Mostrar el contenido de esos 4 bytes en el caso de que:
  - a) Se almacene el entero de 4 bytes 0.
  - b) Se almacene el entero de 4 bytes 1.
  - c) Se almacene el entero de 4 bytes -1.
  - d) Se almacenen dos enteros de 2 bytes: el 0 y el 65535.
  - e) Se almacenen dos enteros de 2 bytes: el 65535 y el 0.
  - f) Se almacene el número real de 4 bytes 1.0.
  - g) Se almacene el número real de 4 bytes -1.0.

Para mostrar el contenido, defina una función que recibe un objeto y escribe en la salida estándar las distintas interpretaciones: el entero, el flotante, los dos enteros de 16 bits, y los 4 enteros de 8 bits. Adicionalmente, deberá imprimir los 32 bits desde el más significativo al menos significativo.

# 2

# Gestión de proyectos



Introducción .....	9
Problema: círculo central	
Compilación y ejecución	
Módulos y compilación separada .....	10
Separación en archivos	
Compilación, enlazado y ejecución	
Bibliotecas	
Gestión del proyecto con make.....	15
Dependencias entre módulos	
Archivo <i>makefile</i>	
Reglas implícitas	
Mejora y ampliación de la biblioteca .....	22
Programas a desarrollar	
Práctica a entregar	

## 2.1 Introducción

Este capítulo se presenta como un guión práctico guiado, pero a la vez incluyendo los comentarios y explicaciones necesarias para que sirva como referencia básica en la creación y gestión de proyectos con archivos *makefile*. Los objetivos de este capítulo son los siguientes:

1. Crear un programa usando compilación separada con el compilador *g++* de la *GNU*.
2. Crear y usar bibliotecas. Conocer la orden *ar* y el uso de bibliotecas con *g++*.
3. Aprender a manejar archivos *makefile* básicos. Entender cómo se puede gestionar automáticamente la compilación con la orden *make*.
4. Practicar con el uso de funciones generales y estructuras.
5. Introducirse en la idea del mantenimiento de un módulo software.

Los requisitos para poder realizar esta práctica son:

1. Saber dividir un programa en distintos módulos, creando archivos de cabecera (“*.h*”) y compilables (“*.cpp*”).
2. Conocimientos básicos sobre funciones, incluyendo el paso por valor y por referencia.
3. Saber manejar estructuras (*struct*).
4. Conocimientos básicos de E/S de texto.

El problema a resolver se ha intentado crear de una forma bastante simplificada para poder centrar la atención en los contenidos referidos a la compilación separada y la gestión de proyectos con *make*. Cuando avance en sus conocimientos de programación en C++, podrá comprobar que hay otras formas de definir nuevos tipos de datos, especialmente atendiendo al problema de ocultamiento de la representación que garantiza que los objetos siempre contienen datos válidos.

### 2.1.1 Problema: círculo central

Para desarrollar este guión práctico nos basaremos en un problema muy simple: el cálculo de un círculo central. El problema y la solución propuesta están diseñados para enfatizar los distintos aspectos que queremos dejar claros. Lógicamente, el lector podría haber propuesto otra solución para el mismo problema.

El problema consiste en leer dos círculos de entrada y calcular el círculo central que pasa por el centro de ambos. Para ello creamos un programa que incluye dos nuevos tipos de datos:

1. El tipo *Punto* que contiene dos objetos, de tipo *double*, para las coordenadas *x* e *y* correspondientes. Un objeto de este tipo representa un punto (*x,y*) en el espacio 2D.
2. El tipo *Circulo* que está compuesto de dos objetos: uno de tipo *Punto* para indicar el *centro* y otro de tipo *double* para indicar el *radio*. Un objeto de este tipo representa un círculo *radio*-(*x,y*) en el espacio 2D.

Además, junto con estos tipos, se crean un conjunto de funciones que realizan operaciones con ellos. Por ejemplo, podemos resolver:

- E/S de un punto o un círculo desde/hacia la entrada/salida estándar.
- Dados dos puntos, cuál es la distancia euclídea entre ellos.
- Dados dos puntos, cuál es el punto medio entre ellos.
- Dado un punto y un círculo, determinar si el punto está en el interior del círculo.
- Dado un círculo, obtener el área del círculo.
- Etc.

Con estas funciones y los nuevos tipos, nos resultará bastante simple resolver el problema del círculo central.

## 2.1.2 Compilación y ejecución

Inicialmente, podemos tener el programa escrito completamente en un mismo archivo: por ejemplo, `central.cpp`. Por tanto, en éste estarán las definiciones de los tipos, las funciones que operan con ellos y la función `main` que resuelve el problema del círculo central.

La compilación del programa se puede realizar de una forma muy sencilla en una única línea, indicando al compilador que obtenga el ejecutable correspondiente:

```
prompt> g++ -o central central.cpp
```

Observe que hemos llamado al compilador —`g++`— indicando dos cosas:

1. El nombre del archivo resultante, con la opción `-o`.
2. El nombre del archivo a compilar, es decir, el código fuente `central.cpp`.

En principio, el orden de los argumentos que se pasan al programa `g++` no es importante, se puede cambiar. Ahora bien, es importante tener en cuenta que el nombre del resultado vendrá, obligatoriamente, después de la opción `-o`. En caso de no hacerlo, por ejemplo, intercambiando el orden de `central` y `central.cpp`, el compilador podría entender que el resultado tiene que grabarse sobre el archivo `central.cpp`, y un intento de sobreescribir este archivo puede llevarnos a perder el código fuente del programa.

La compilación y ejecución, por tanto, podrían realizarse con las siguientes líneas:

```
prompt> g++ -o central central.cpp
prompt> ./central
Introduzca un círculo en formato radio-(x,y): 3-(0,0)
Introduzca otro círculo: 4-(5,0)
El círculo que pasa por los dos centros es: 2.5-(2.5,0)
```

En la primera línea hemos compilado y obtenido un ejecutable válido, ya que no se ha generado ningún mensaje de error. En la segunda, hemos lanzado el programa indicando el nombre del archivo ejecutable e indicando que se encuentra en el directorio actual. En la tercera y cuarta hemos introducido dos círculos, y en la última línea nos ha escrito el resultado, un círculo de radio 2.5 centrado en (2.5,0).

**Ejercicio 2.1 — Compilar y ejecutar.** Complete el programa `central.cpp` y ejecútelo para comprobar su correcto funcionamiento.

## 2.2 Módulos y compilación separada

En el problema que hemos resuelto hemos creado dos nuevos tipos de datos, `Punto` y `Circulo`, así como una serie de operaciones para ellos. Con esas utilidades, resulta más sencillo resolver problemas que requieran de puntos y círculos. Por ejemplo, si ahora queremos crear un nuevo programa que pregunte por un círculo y que escriba el área correspondiente, no tenemos más que crear un nuevo `main` donde se use el tipo `Circulo` y sus operaciones. Sin embargo, resulta incómodo tener que crear un archivo `area.cpp`, donde copiar/pegar todo el código de puntos y círculos, para finalmente añadirle una función `main` que resuelve este problema.

La creación de una solución basada en módulos independientes, que se compilan de forma separada, nos facilita en gran medida la reutilización de dichos módulos en nuevos programas. En nuestra solución anterior hemos obtenido:

1. Una definición de `Punto`, y una serie de operaciones con este tipo. Este conjunto de utilidades se puede considerar como una unidad, un módulo que compone nuestra solución.
2. La definición de `Circulo` y las operaciones con este tipo. Este módulo hace uso del anterior para definir el centro del círculo.
3. Un módulo para calcular el círculo central. Este módulo usa los dos anteriores en una función `main` que resuelve un problema concreto.

Si hacemos que los tres módulos se escriban de forma independiente, haremos más sencillo crear programas que trabajen con puntos y círculos. Por ejemplo, si ahora queremos obtener el programa `area`, sólo es necesario crear un `main` para este programa e indicar que el programa lo componen este módulo junto con los dos anteriores. Si queremos obtener un programa que nos indique la distancia entre dos puntos, podemos crear una función `main` en un archivo `distancia.cpp` e indicar que el programa está compuesto por este módulo junto con el módulo `Punto`.

## 2.2.1 Separación en archivos

Para generar los distintos módulos de forma independiente, crearemos distintos archivos fuente —.cpp— para separar cada una de las partes. Concretamente:

1. Fichero `punto.cpp`. En este archivo se incluye todo lo relacionado con puntos. Por tanto, debe incluir la estructura `Punto`, junto con las operaciones correspondientes.
2. Fichero `circulo.cpp`. En este archivo se incluye todo lo relacionado con círculos. Por tanto, debe incluir la estructura `Circulo`, junto con las operaciones correspondientes.
3. Fichero `central.cpp`. En este archivo se incluye la función `main` que implementa el algoritmo de cálculo del círculo central.

Para que esta división sea correcta, será necesario crear dos archivos cabecera para los dos primeros módulos:

1. Fichero `punto.h`. Este archivo contiene la definición del tipo `Punto` y todas las cabeceras de las funciones del módulo. Note que el archivo `punto.cpp` no contiene directamente esta estructura, ya que en realidad lo que hace es incluir `punto.h`. Cuando queramos que un programa use el tipo `Punto` y sus operaciones, no tendremos más que incluir (con `#include`) este archivo cabecera.
2. Fichero `circulo.h`. Este archivo contiene la definición del tipo `Circulo` y todas las cabeceras de las funciones del módulo. Al igual que el anterior, el `cpp` no contiene directamente la estructura, sino que la incluye desde este archivo cabecera. Note que este archivo siempre va a requerir del tipo `Punto` antes que él, ya que es necesario para definir el tipo `Circulo`. Por tanto, habrá una línea `#include` para incluir el archivo `punto.h` antes de definir `Circulo`.

En el archivo cabecera incluimos todo el código necesario para poder usar las herramientas que “exporta” un módulo. Si nuestro programa usa puntos y círculos y queremos usar los dos módulos, tendremos que incluir ambos archivos cabecera.

Algunas preguntas habituales para poder realizar esta separación son:

1. ¿En cuántos sitios aparece la estructura `Punto` y `Círculo`? Como norma general, en nuestros programas cualquier componente se definirá en un único sitio. En este caso estas estructuras se necesitan en varios sitios, pero realmente sólo se han definido en los archivos `punto.h` y `circulo.h`, respectivamente.
2. ¿Qué `includes` pongo en los archivos `.h`? Aunque tengamos varios archivos, la forma de decidir los archivos que se incluyen no cambia. Si está editando un archivo concreto —archivo `X.h` o `X.cpp`— lo único que tiene que decidir es si éste archivo que está editando necesita del archivo cabecera o no. Por ejemplo:
  - Si edita `punto.h`, puede pensar en añadir la cabecera `iostream`, pero si observa el contenido de este archivo no hay nada en él que necesite de los recursos de `iostream` y por tanto no debería incluirlo. Lógicamente, en `punto.cpp` si lo necesitará, por ejemplo, para usar `cin` o `cout`.
  - Si edita `circulo.h`, tendrá una situación similar, aunque si observa la definición del tipo se dará cuenta que se usa el tipo `Punto`, por tanto, este archivo necesita conocer esa estructura, es decir, necesitará incluir el archivo `punto.h`.
3. ¿En qué directorios guardo los 5 archivos generados? Por ahora, todos los archivos se almacenan en el mismo directorio. Esto nos permite facilitar la inclusión de los archivos cabecera. En este sentido, no se olvide de incluir los archivos de su proyecto con las comillas dobles. Por ejemplo, en lugar de hacer `#include<punto.h>`, debería hacer `#include "punto.h"`. Más adelante se volverá sobre este asunto.
4. ¿Dónde ponemos el `using namespace std`? Esta discusión se sale de los objetivos de este guión, y se dejará para más adelante. En cualquier caso, tenga en cuenta que esa línea nos sirve para poder usar más cómodamente los recursos del estándar, y será suficiente con ponerla al comienzo de los archivos `cpp`.

**Ejercicio 2.2 — Separar archivos.** A partir del programa `central.cpp`, cree el conjunto de 5 ficheros que componen el resultado de separar los tres módulos.

### Inclusión múltiple de un mismo archivo cabecera

Antes de compilar y depurar el programa asegúrese de que:

1. El archivo `punto.cpp` incluye el archivo `punto.h`.
2. El archivo `circulo.cpp` sólo incluye el archivo `circulo.h`. Recuerde que éste archivo cabecera ya incluye el archivo `punto.h`.
3. El archivo `central.cpp` sólo incluye el archivo `circulo.h`. Recuerde que éste archivo cabecera ya incluye el archivo `punto.h`.

Una vez disponemos de los 5 archivos, podemos obtener el ejecutable con la siguiente línea:

Consola

```
prompt> g++ -o central punto.cpp circulo.cpp central.cpp
```

que estudiaremos en más detalle en la siguiente sección. Ahora, simplemente use esa línea para intentar compilar y depurar los errores que haya podido cometer.

Observe que se ha especificado la inclusión de los archivos cabecera necesarios, pero teniendo en cuenta la inclusión indirecta del archivo `punto.h`. En la práctica, esta forma de trabajo no tiene sentido, ya que cuando estamos editando un archivo fuente y necesitamos recursos de un archivo cabecera, simplemente lo incluimos sin tener en cuenta si se incluyen otras cosas indirectamente.

Para provocar una situación más realista, supongamos que no sabemos qué inclusiones indirectas podemos encontrar en los archivos cabecera. Cuando desarrollamos el archivo `central.cpp`, podemos considerar que necesitaremos los recursos que

nos ofrecen los dos archivos cabecera: `punto.h` y `circulo.h`. Por tanto, podemos incluirlos en nuestro programa, por lo que el resultado sería que nuestro archivo `central.cpp` termina incluyendo dos veces el archivo `punto.h`, una directamente y otra indirectamente a través de `circulo.h`.

**Ejercicio 2.3 — Inclusión múltiple.** Incluya también el archivo `punto.h` en el archivo `central.cpp`. Compruebe que obtenemos un error de compilación.

El problema que nos encontramos es que la división del programa en múltiples ficheros puede provocar que una misma estructura termine incluyéndose varias veces en una misma *unidad de compilación*, es decir, en un archivo `cpp` aparece dos veces la misma definición.

En principio podríamos pensar que no debería ser problema, ya que es una definición idéntica; sin embargo, el lenguaje especifica que esa definición múltiple no es válida. Para resolverlo sólo tenemos una solución: evitar que se compile dos veces un mismo archivo cabecera.

Para evitar la inclusión múltiple de un archivo hacemos uso del *precompilador*. Éste realiza una primera etapa de precompilación sobre el fuente para obtener un fichero procesado listo para el compilador. Esta etapa de precompilación realmente realiza operaciones relativamente sencillas, operaciones que se indican con las directivas de precompilación, con líneas que comienzan con el carácter '`#`'. Un ejemplo es la directiva `#include`, que como hemos visto, inserta el contenido de un fichero en otro.

Nuestros archivos cabecera usarán las directivas `#ifndef`/`#endif` y `#define` para establecer un sistema que evita la inclusión múltiple. El esquema de cualquier archivo cabecera será el siguiente:

```
#ifndef _ARCHIVO_CABECERA_H
#define _ARCHIVO_CABECERA_H

// < contenido del archivo cabecera >

#endif
```

donde hemos creado —*inventado*— un nombre especialmente para este archivo cabecera: `_ARCHIVO_CABECERA_H`. Este nombre debe ser único para un archivo, por lo que será necesario crear un nombre nuevo por cada archivo cabecera nuevo. Lo más simple, es crear un nombre asociado al nombre del archivo.

La forma en que funciona este trozo de código es muy simple. El precompilador encuentra `#ifndef`, por lo que recibe la orden de compilar el código que viene a continuación sólo en el caso en que no esté definido ese identificador. Observe que la primera vez que se encuentre el archivo cabecera, este identificador no está definido, pero al tener la definición dentro, la segunda vez que se lo encuentra sí lo estará. Imagine que hemos creado este esquema para el fichero `punto.h`, que incluye la estructura `Punto`. Si hay doble inclusión del archivo cabecera, el precompilador encuentra algo parecido a:

```
#ifndef _PUNTO_H_
#define _PUNTO_H_

    struct Punto {
        double x,y;
    };
    // ... y otras cosas ...

#endif

#ifndef _PUNTO_H_
#define _PUNTO_H_

    struct Punto {
        double x,y;
    };
    // ... y otras cosas ...

#endif
```

donde hemos eliminado algunos detalles del contenido para centrarnos en el problema que surge al tener la doble definición del tipo `Punto`.

En esta doble inclusión, el precompilador recorre el código de arriba hacia abajo, pasando por el primer `#ifndef`, entrando en él y definiendo la constante con `#define`. Cuando se encuentra con el segundo `#ifndef`, ya habrá pasado por el primero, habrá definido la constante `_PUNTO_H_` y no tendrá en cuenta el nuevo código repetido.

Por tanto, a partir de este momento, siempre que cree un archivo cabecera, añada las líneas necesarias para proteger la inclusión múltiple. Observe que las líneas `#ifndef`/`#define` estarán antes del código del archivo —incluso de los “`#includes`”— y la línea `#endif` será la última.

**Ejercicio 2.4 — Completar cabeceras.** Modifique los archivos cabecera para protegerlos de la inclusión múltiple. Compruebe que funciona correctamente en el caso en que se incluyen los dos en el archivo `central.cpp`.

## 2.2.2 Compilación, enlazado y ejecución

Para realizar la compilación y obtener el ejecutable, podemos usar la siguiente orden:

```
prompt> g++ -o central punto.cpp circulo.cpp central.cpp
```

En este caso, hemos escrito una línea similar a cuando teníamos un único archivo, pero especificando los tres. El proceso para obtener el archivo ejecutable realmente es más complejo, ya que es necesario:

1. Compilar cada uno de los tres archivos. Es decir, obtener un fichero objeto —traducción— desde cada uno de los archivos fuente `cpp`.
2. Enlazar los tres ficheros objeto obtenidos en la etapa anterior para crear el ejecutable. En este caso no hay que traducir, sino enlazar. Por ejemplo, en el programa central se llama a la función de lectura de un punto, por lo que en el ejecutable este punto de llamada debe quedar “enlazado” al punto donde está definida dicha función.

Como resultado podemos obtener distintos tipos de errores: errores de compilación en un archivo (nos indicará el archivo y el punto donde encuentra algún problema) o errores de enlazado (nos indicará la función o símbolo que no encuentra o resuelve). Si no indica nada, es que ha obtenido el ejecutable sin problemas.

La compilación y enlazado directo se debe a que el programa `g++` nos facilita la operación, puesto que sabe distinguir el tipo de archivos que le damos como entrada. Como queremos obtener un ejecutable y le damos tres archivos fuente, internamente realiza las tres traducciones a archivos objeto así como la llamada al enlazador para obtener el resultado final.

Nosotros estamos interesados en detallar cada uno de los pasos que se deben realizar. Por lo tanto, vamos a realizar la misma operación paso a paso:

```
prompt> g++ -c -o central.o central.cpp
prompt> g++ -c -o punto.o punto.cpp
prompt> g++ -c -o circulo.o circulo.cpp
prompt> g++ -o central punto.o circulo.o central.o
```

Con estas cuatro líneas, hemos obtenido cuatro archivos nuevos:

1. Tres archivos objeto `.o`— en las tres primeras líneas.
2. Un ejecutable —`central`— en la última línea.

En estas órdenes podemos distinguir:

- En la compilación se indica el archivo fuente y el objeto que vamos a obtener. Como sabemos, la opción `-o` precede al nombre del archivo resultado. Le asignamos esa extensión para distinguirlo como *archivo objeto*.
- En la compilación se indica la opción `-c`. Con esta opción el compilador sabe que se tiene que limitar a compilar, es decir, traducir en un archivo objeto que no es ejecutable.
- En la última línea, no existe la opción `-c`. Cuando no existe, `g++` entiende que la intención del usuario es obtener un ejecutable, por lo que realizará lo necesario para obtener el ejecutable. En nuestro caso, simplemente *enlazar*.
- Se compilan archivos `cpp`. Como era de esperar, los archivos cabecera `.h`— sólo existen para ser insertados en los archivos `cpp` a través de la directiva `#include`. Podríamos decir que no son más que “trozos” de código C++ que se insertan en los archivos `cpp` donde se compilarán como parte de un fichero más complejo.

**Ejercicio 2.5 — Compilar, enlazar y ejecutar.** Realice la compilación y enlazado del programa a partir de los 3 archivos fuente que hemos indicado. Ejecute para comprobar su correcto funcionamiento.

## 2.2.3 Bibliotecas

El problema que hemos resuelto ha dado lugar a dos módulos reutilizables: *Punto* y *Circulo*. En la práctica, podemos crear programas que usen las herramientas contenidas en uno o en los dos módulos. Por ejemplo:

1. *Distancia entre puntos*. Para crear un programa que obtenga la distancia entre dos puntos, se puede obtener el ejecutable creando un archivo con un `main` y enlazándolo con el módulo *Punto*.
2. *Área*. Para crear un programa que lee un círculo y obtiene el área del círculo, podemos crear el ejecutable con un `main` que se enlaza con los dos módulos *Punto* y *Circulo*.

En el primer caso habrá que enlazar con un solo módulo, en el segundo con los dos. En la práctica es posible crear módulos reutilizables mucho más numerosos y complejos. Por ejemplo, imagine que creamos un conjunto de módulos para operar con formas del espacio 2D: puntos, círculos, rectángulos, triángulos, etc.; podemos obtener un número bastante grande de módulos que tendremos que gestionar para poder crear programas que usen alguno o varios de dichos módulos.

La gestión de un grupo de módulos relacionados se puede simplificar por medio de las bibliotecas<sup>1</sup>. Una biblioteca no es más que un grupo de módulos compilados y empaquetados en un mismo archivo. Podríamos decir que es un contenedor de archivos objeto (`.o`).

La ventaja es que cuando usemos bibliotecas no será necesario indicar todos y cada uno de los módulos objeto que hacen falta para obtener el ejecutable. En lugar de eso, basta con indicar el nombre de la biblioteca; el enlazador se encarga de

<sup>1</sup>En inglés “library”. Está muy extendido el uso de la palabra “librería” en lugar de biblioteca debido a una mala traducción del inglés.

extraer y añadir los módulos que hagan falta para el programa ejecutable. Observe que no es necesario añadirlos todos, sino que sólo se usarán los que el programa requiera.

Por ejemplo, si tenemos una biblioteca con 100 archivos objeto empaquetados, es posible que el programa use las herramientas de uno solo de ellos, y por tanto, el enlazador sólo extraiga y añada ese módulo. Es más, incluso si indicamos que enlace con una biblioteca, podría no extraer ninguno si no fuera realmente necesaria.

Para crear archivos biblioteca será necesario usar la orden `ar` (de “archive”). La forma en que vamos a usar esta orden se puede indicar como sigue:

```
ar <operación> <biblioteca> [<archivos>]
```

donde podemos distinguir tres partes:

1. *Operación*. En general, es una letra que indica lo que queremos realizar. Por ejemplo, podemos añadir archivos, reemplazar, consultar, etc.
2. *Biblioteca*. Es el nombre del archivo biblioteca con el que queremos trabajar.
3. *Archivos*. Podemos indicar cero o más archivos objeto con los que realizar la operación.

Aunque existen múltiples posibilidades, nosotros vamos a usar esta orden de una manera muy concreta y simple, de forma que todo lo que vamos a necesitar se puede realizar con una orden:

1. La operación la especificaremos siempre como `rvs`. La operación propiamente dicha es `r`, es decir, insertar módulos con reemplazo. Las letras `vs` se indican para obtener información de lo que se hace —la primera— y para que se añada o actualice un índice con los contenidos del archivo —la segunda—.
2. La biblioteca será el nombre del archivo con el que trabajar. Si el nombre existe, la operación de inserción se encarga, además, de crear el archivo.
3. El módulo o módulos a incluir en la biblioteca. Si no están ya, se insertarán como nuevos; si ya existen, se reemplazan.

En nuestro ejemplo de puntos y círculos, hemos creado dos módulos con los que enlazar nuestros programas. Podemos crear una biblioteca con la siguiente línea:

```
Consola
prompt> ar rvs libformas.a punto.o circulo.o
```

Observe el nombre que hemos creado para la biblioteca. El primer lugar, tiene una extensión `.a` (de “archive”). Por otro lado, empieza por `lib` (de “library”). Todos los nombres tendrán esta estructura:

```
lib<nombre>.a
```

Por tanto, la biblioteca que hemos creado se llama `formas`. Es interesante destacar que si ahora modificamos uno de los archivos objeto, por ejemplo, porque hemos cambiado el `cpp` y lo recompilamos, podemos ejecutar exactamente la misma orden para obtener la biblioteca actualizada.

### Enlazar con bibliotecas

La forma directa y simple para comprobar que nuestra nueva biblioteca funciona correctamente y nos permite obtener el ejecutable deseado es indicando el nombre de ésta en lugar de los archivos objeto. Es decir, enlazar con la siguiente orden:

```
Consola
prompt> g++ -o central central.o libformas.a
```

Con lo que obtendríamos exactamente el mismo ejecutable. Sin embargo, no es la forma habitual de usar las bibliotecas. Normalmente se usa la opción `-l` de `g++` para indicar una biblioteca con la que enlazar. La línea podría ser la siguiente:

```
Consola
prompt> g++ -o central central.o -lformas
```

Observe que la opción va seguida del nombre de la biblioteca, no del archivo. Si ejecuta esta línea es probable que obtenga un error de enlazado, ya que nos indica que no encuentra la biblioteca `formas`. El problema es que cuando decimos de enlazar con una biblioteca, el enlazador tiene un conjunto de directorios muy concretos donde encontrar las bibliotecas. Lógicamente, en principio, sólo contiene algunos directorios del sistema donde se encuentran las bibliotecas que se hayan instalado y no el directorio actual.

La solución al problema es sencilla, pues no tenemos más que incluir un nuevo directorio donde buscar bibliotecas. La biblioteca `formas` está creada en el directorio donde estamos trabajando, así que si incluimos el directorio actual para buscarla, el enlazador podrá obtener el resultado que queremos. El directorio donde trabajamos se puede indicar con un punto “.” por lo que basta con añadir este directorio por medio de la opción `-L` (en mayúscula). En concreto, la línea sería la siguiente:

```
prompt> g++ -o central central.o -L. -lformas
```

**Ejercicio 2.6 — Crear y usar una biblioteca.** Use la orden `ar` para crear una biblioteca `formas` y vuelva a generar el ejecutable enlazando con ella.

Finalmente, es importante enfatizar el resultado que hemos obtenido. La biblioteca no es más que una colección de archivos objeto que se puede enlazar para obtener programas ejecutables. Tal vez piense que si crea una biblioteca, podrá distribuirla a sus colegas entregando el archivo de extensión `.a`, pues tiene los archivos compilados. Evitaría de esta forma revelar el código fuente de sus programas. No es así.

Si queremos obtener nuevos programas, no sólo necesitamos la biblioteca, sino los archivos cabecera que deberán incluirse para acceder a las utilidades que ofrece. Por ejemplo, aunque tengamos el archivo `libformas.a`, nunca podremos obtener el archivo `central.o` si no disponemos de los dos archivos cabecera (`punto.h` y `circulo.h`). Por esta razón, cuando se distribuye una biblioteca para que los programadores desarrollen nuevos programas, seguramente tendrá un grupo de archivos que incluye archivos cabecera.

### Orden de bibliotecas

Inicialmente hemos presentado la orden `g++` indicando que el orden de los parámetros no es relevante, pues podemos cambiarlo obteniendo el mismo resultado. Sin embargo, las bibliotecas que aparecen en una línea deben estar correctamente ordenadas.

Básicamente, el enlazador pasa por cada una de las bibliotecas de izquierda a derecha, una sola vez, extrayendo y añadiendo lo que se necesita. Para decidir lo que necesita, debe revisar el conjunto de símbolos sin resolver —es decir, sin enlazar— que aún están pendientes.

Por ejemplo, si en el programa tenemos una llamada a una función `Distancia`, al pasar por las bibliotecas consulta si esta función está incluida en algún módulo objeto, en cuyo caso este módulo se añade al ejecutable. Ahora bien, una vez añadida esta función, se pueden haber creado nuevos símbolos que resolver, símbolos que esta función llama. Estos nuevos símbolos se buscarán en las siguientes bibliotecas y no en las ya revisadas.

## 2.3 Gestión del proyecto con make

En las secciones anteriores se han presentado las órdenes necesarias para compilar un proyecto con múltiples archivos, incluyendo una biblioteca y un ejecutable. Estas órdenes se ejecutan una vez están terminados todos los archivos fuente. Sin embargo, en la práctica, es normal que se estén desarrollando los archivos fuente mientras se están recompilando los programas, ya sea para corregir errores de compilación, para modificar el programa, para ampliarlo, etc.

La separación de la solución en módulos independientes facilita y hace más eficiente esta forma de trabajo. Por ejemplo, si desarrollamos el módulo `punto.cpp` y lo compilamos, ya no será necesario volver a compilarlo mientras no lo modificuemos. Por ejemplo, si una vez que tenemos un ejecutable modificamos una línea del archivo `central.cpp`, sólo será necesario volver a compilar este módulo y enlazar el resultado —con la misma biblioteca— para actualizar el ejecutable.

El problema surge cuando realizamos un cambio en una parte del proyecto que afecta a más módulos, especialmente si tenemos un proyecto mucho más complejo. En este caso, puede ser muy complicado determinar los módulos que tenemos que volver a compilar, lo que nos puede llevar a tener que recompilarlo todo para asegurarnos de que el ejecutable se actualiza correctamente.

La orden `make` nos permite gestionar todos los archivos de un proyecto y ayudarnos a realizar las actualizaciones necesarias para generar los archivos ejecutables.

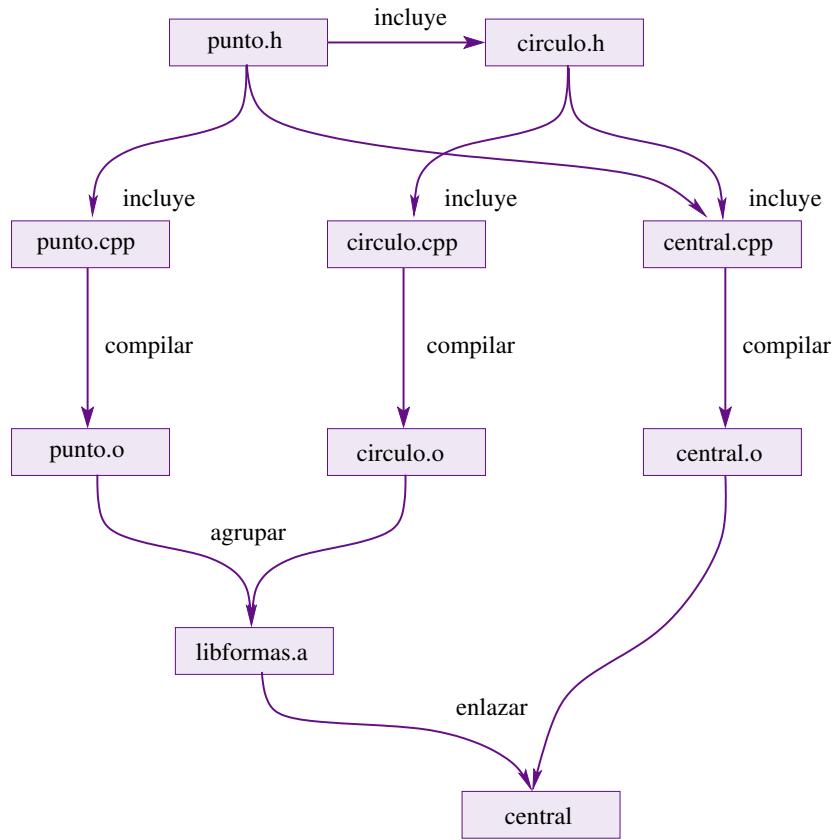
### 2.3.1 Dependencias entre módulos

Si analizamos los módulos que hemos generado en las secciones anteriores y los representamos gráficamente, podemos obtener el gráfico de la figura 2.1.

En este gráfico de dependencias se refleja claramente cómo podemos llegar desde los archivos fuente hasta el archivo ejecutable `central`. Por tanto, conociendo estas dependencias junto con las órdenes de compilación que hemos estudiado, podemos ser capaces de regenerar el ejecutable cuando realicemos alguna modificación. Observe que:

- Las etiquetas `incluye` corresponden a directivas `#include` del código. El archivo `central.cpp` incluye los dos cabeceras, aunque podría haberse creado incluyendo sólo el archivo `circulo.h`. Si se hubiera incluido sólo éste, podríamos eliminar una línea de inclusión desde `punto.h` al `central.cpp`. Sin embargo, note que todavía se podría observar que el archivo `punto.h` se incluye indirectamente al incluir `circulo.h`.
- Las etiquetas `compilar`, `agrupar` y `enlazar` nos indican que esos módulos se obtienen a partir de los anteriores con los comandos `g++ -c`, `ar`, y `g++` respectivamente.

A partir de este gráfico podemos determinar el conjunto de acciones necesarias cuando se modifica uno de los 5 archivos fuente. Por ejemplo, si modificamos `central.cpp`, podemos ver qué será necesario:



**Figura 2.1**  
Módulos y relaciones para el programa *central*.

1. Regenerar *central.o*. Puesto que el programa es distinto, hay que volver a compilar.
2. Regenerar el ejecutable *central*. Al haber modificado el módulo objeto, hay que volver a obtener el ejecutable: hay que volver a enlazar.

En este supuesto no hemos tenido que modificar ningún otro módulo. En concreto, la biblioteca que se usa en el enlazado no se ha cambiado, ya que el código que se incluye para compilarla es exactamente el mismo.

Podemos analizar un ejemplo más complejo: modificar *circulo.h*. Como sabe, este módulo no se compila directamente, sino que se incluye con la directiva `#include`. Eso significa que los dos módulos compilables *circulo.cpp* y *central.cpp*, a pesar de no haberse modificado directamente, se ven afectados. Cuando se compilan, recuerde que se inserta el código que haya escrito en el fichero *circulo.h*; si modificamos este archivo, estamos modificando el código que llega al compilador desde los *cpp*. Será necesario:

1. Regenerar *central.o*. Hay que volver a compilar.
2. Regenerar *circulo.o*. Hay que volver a compilar.
3. Regenerar *libformas.a*. El módulo *circulo.o* que tenía antes ya no es válido, pues se ha obtenido uno nuevo. Hay que reemplazar ese módulo en la biblioteca.
4. Regenerar el ejecutable *central*. Hay que volver a enlazar.

Podemos hacer nuevos supuestos que implicarían las correspondientes actualizaciones. Por ejemplo, fíjese que si modificamos el archivo *punto.h*, será necesario rehacerlo todo.

La orden `make` nos permitirá gestionar todas estas dependencias de forma automática. En lugar de estudiar los pasos que son necesarios para rehacer el ejecutable, usaremos la orden `make` para que estudie las dependencias que “*han fallado*”, y lance automáticamente cada uno de los pasos para regenerar el archivo ejecutable.

### 2.3.2 Archivo *makefile*

La orden `make` necesita conocer el conjunto de módulos, las dependencias entre ellos y la forma de generarlos para poder gestionar un proyecto de programación. Un archivo *makefile* es un archivo de texto que contiene esta información en un formato que la orden `make` sabe interpretar.

Este tipo de archivo lo nombramos así porque la mayoría de las veces será un archivo de texto que tenga exactamente este nombre (o con la primera letra mayúscula). En general, el nombre de un archivo que contiene información para `make` no tiene por qué llamarse así, de hecho puede ser cualquier nombre, aunque nosotros siempre usaremos este para facilitar su uso.

## Reglas

El contenido del archivo **makefile** permite representar la información que hemos mostrado en el esquema de la figura 2.1. Es decir, permite indicar cuáles son los módulos, cuáles son las dependencias, y cuáles son los comandos necesarios para regenerarlos.

Aunque el gráfico parece muy complejo ya que incluye múltiples dependencias, directas e indirectas, en la práctica es bastante simple especificarlo en forma de texto, ya que lo que hay que incluir en el archivo **makefile** es cada una de las dependencias básicas del proyecto, es decir, las que regeneran los archivos resultado. Será el programa **make** el encargado de “*encadenarlas*”. Concretamente, en nuestro ejemplo habrá que especificar:

1. El archivo **punto.o** se debe crear con una orden **g++ -c** siempre y cuando se hayan modificado los archivos **punto.h** o **punto.cpp**.
2. El archivo **circulo.o** se debe crear con una orden **g++ -c** siempre y cuando se hayan modificado los archivos **punto.h**, **circulo.h** o **circulo.cpp**.
3. El archivo **central.o** se debe crear con una orden **g++ -c** siempre y cuando se hayan modificado los archivos **punto.h**, **circulo.h** o **central.cpp**.
4. El archivo **libformas.a** se debe crear con una orden **ar** siempre y cuando se hayan modificado los archivos **punto.o** o **circulo.o**.
5. El archivo **central** se debe crear con una orden **g++** siempre y cuando se hayan modificado los archivos **central.o** o **libformas.a**.

Observe que aunque los ítems son independientes, es fácil ver que están interrelacionados. Por ejemplo, podemos deducir que si modificamos el archivo **central.cpp**, el punto 3 indica que será necesario crear el archivo **central.o**, pero al modificar éste, vemos que el punto 5 nos indica que debemos generar el archivo **central**.

Esta información es la que vamos a incluir —con un formato muy concreto— en un archivo **makefile**. La orden **make** es la encargada de interpretar estas instrucciones para lanzar todas las órdenes necesarias.

¿Cómo puede saber **make** que un archivo se ha modificado y hay que volver a regenerar un módulo? Simplemente comprobando la fecha/hora de última modificación del archivo. Recuerde que en el disco, junto a los ficheros, se almacenan la fecha y hora de última modificación. Si comprobamos las fechas de **central.o** y **central.cpp**, podemos determinar que **central.o** hay que regenerarlo en caso de que su fecha sea anterior a la de **central.cpp**.

La estructura básica para representar esta información es una **regla**:

**Objetivo** : *Lista de dependencias*  
 ⇒ **Acciones**

En donde:

- **Objetivo**: Indica lo que queremos construir. Por ejemplo, el módulo **libformas.a**.
- **Lista de dependencias**: Esto es una lista de ítems de los que depende la construcción del objetivo de la regla. Al dar esta lista de dependencias, la utilidad **make** debe asegurarse de que han sido satisfechas antes de poder alcanzar el objetivo de la regla. Por ejemplo, **libformas.a** es un objetivo que depende de **punto.o** y **circulo.o**.
- **Acciones**: Este es el conjunto de acciones que se deben llevar a cabo para conseguir el objetivo. Normalmente serán instrucciones como las que hemos visto antes para compilar, enlazar, etc. Por ejemplo, el objetivo **libformas.a** se consigue lanzando una orden **ar rvs libformas.a punto.o circulo.o**.

Por tanto, en nuestro problema usaremos una regla para codificar como objetivo cada uno de los 5 ficheros que queremos generar. Además, aunque se escriban como reglas independientes, la orden **make** sabe relacionar unas con otras, ya que las dependencias de unas reglas son los objetivos de otras.

Recuerde que en el gráfico de la figura 2.1 hemos presentado los archivos que se generan, las dependencias, y las acciones para regenerarlos. En nuestro ejemplo podemos escribir las siguientes reglas para reflejar esa información:

```

1 punto.o : punto.h punto.cpp
2         g++ -c -o punto.o punto.cpp
3 circulo.o : punto.h circulo.h circulo.cpp
4         g++ -c -o circulo.o circulo.cpp
5 central.o : punto.h circulo.h central.cpp
6         g++ -c -o central.o central.cpp
7 central : central.o libformas.a
8         g++ -o central central.o -L. -lformas
9 libformas.a : punto.o circulo.o
10        ar rvs libformas.a punto.o circulo.o

```

Este conjunto de reglas se pueden almacenar en un archivo de texto con nombre **Makefile**, en el mismo directorio donde tenemos los archivos fuente del programa. Después de ello, podemos procesarlo con la siguiente orden:



Consola

prompt> make central

En la que se ha indicado a `make` que intente obtener el objetivo `central`. No es necesario indicarle el archivo con las reglas, ya que al no especificar nada, buscará automáticamente un archivo con el nombre `makefile`<sup>2</sup> (o `Makefile`).

Respecto a la sintaxis usada para escribir estas reglas en el archivo `makefile`, es muy importante tener en cuenta que:

- El orden de las reglas no afecta al resultado.
- Las acciones —órdenes— que se incluyen en cada regla deben estar precedidas por un tabulador.

El hecho de añadir un tabulador a las acciones es fundamental para que funcione correctamente. Una regla puede tener un número indeterminado de acciones a realizar (una detrás de otra) y la forma de saber si hay más acciones es comprobando si la siguiente línea comienza con un tabulador. En el esquema anterior hemos añadido  $\Rightarrow$  precisamente para enfatizar esta tabulación.

Finalmente, aunque el orden de las reglas no afecte al resultado, es habitual situar la regla con el objetivo final como primera regla. En nuestro ejemplo, la regla del objetivo `central` que habíamos situado en la línea 7 se situaría en primer lugar. Con este orden el uso de `make` se hace aún más simple, ya que si llamamos a `make` sin indicar ningún objetivo, entenderá que nuestro objetivo es el de la primera regla.

**Ejercicio 2.7 — Archivo makefile simple.** Edite un archivo `Makefile` en el directorio de trabajo y compruebe que funciona correctamente. Para ello, haga alguna modificación en un archivo fuente y vuelva a lanzar la orden `make`.

## Macros

En un archivo `makefile` se pueden incluir macros como identificadores que nos sirven para parametrizar el archivo. Su sintaxis es la siguiente:

`<NOMBRE> = <texto correspondiente>`

donde:

- `<NOMBRE>` es el nombre de la macro, y corresponde a un identificador —letras, dígitos y `'_'`— que normalmente se escribe con todas las letras en mayúscula.
- `<texto correspondiente>` es el texto por el que sustituir la macro.

Para usar una macro, simplemente escribimos el nombre entre paréntesis —o llaves— precedido de carácter `'$'`. Existe una lista bastante grande de nombres que se usan habitualmente para parametrizar algunos valores en un archivo `makefile`. Algunas de ellas son:

- `AR`: programa para mantener las bibliotecas.
- `CXX`: programa que se usa para compilar código C++.
- `CXXFLAGS`: opciones —flags— adicionales para añadir al compilador de C++.
- `LDFLAGS`: opciones —flags— adicionales para añadir cuando se invoca al enlazador.
- `LDLIBS`: bibliotecas a usar en la etapa de enlazado.

Podemos definir y usar estas variables en nuestro archivo `makefile`. Por ejemplo, si usamos las que hemos indicado en nuestro ejemplo, podría comenzar como sigue:

```

1 AR= ar
2 CXX = g++
3 CXXFLAGS= -Wall -g
4 LDFLAGS= -L.
5 LDLIBS= -lformas
6
7 punto.o : punto.h punto.cpp
8     $(CXX) -c $(CXXFLAGS) -o punto.o punto.cpp

```

De esta forma, es muy simple realizar un pequeño cambio que afecte a múltiples reglas. Por ejemplo, según este ejemplo hemos añadido la opción `-g` en la compilación, es decir, hemos indicado que queremos obtener un código preparado para ser procesado en el depurador. Cuando tengamos terminado todo el programa y no sea necesario depurar, podemos cambiar esa opción por `-O`, es decir, generar un código que no se puede usar con el depurador, pero que es más eficiente al estar optimizado.

**Ejercicio 2.8 — Incluir macros habituales.** Incluya las macros que se han comentado en su fichero y adapte las reglas para que haga uso de ellas.

## Múltiples objetivos finales

El proyecto que hemos resuelto nos permite obtener un archivo ejecutable —`central`— a partir de un conjunto de fuentes. Al escribir la orden `make` en el terminal obtenemos el ejecutable correspondiente, ya que la regla de este objetivo está en primer lugar. Recordemos que esta orden, sin parámetros, hace que se actualice el primer objetivo, es decir, el de la primera regla.

<sup>2</sup>Si hubiéramos puesto un nombre distinto al fichero con las reglas, tendríamos que haber indicado ese nombre con la opción `-f`.

Si el proyecto es más complejo, es probable que tengamos varios programas a generar. Por ejemplo, imagine que además del ejecutable `central` deseamos también un ejecutable `area` que corresponde a un archivo `area.cpp` que se compila y enlaza con nuestra biblioteca. Al tener dos reglas, una para `central` y otra para `area`, sólo una de ellas puede estar en primer lugar, por lo que escribir `make` sin nada más nos llevaría a obtener sólo uno de los objetivos. Por supuesto, siempre podemos hacer `make central` y `make area`, aunque no es la solución más cómoda, especialmente si hay muchos objetivos a alcanzar.

Para resolver el problema podemos usar un objetivo simbólico, o ficticio, que sólo nos sirve para poder generar ambos ejecutables, sin ser realmente un archivo a generar. En concreto, escribimos una primera regla vacía —sin acciones— con un nombre de objetivo cualquiera —no se va a generar el archivo— y con unas dependencias que corresponden a todos los objetivos que deseamos generar.

Por ejemplo, a continuación mostramos el archivo `makefile` modificado, incluyendo una primera regla “ficticia”:

```

1 all : central area
2
3 central : central.o libformas.a
4         g++ -o central central.o -L. -lformas
5
6 area : area.o libformas.a
7         g++ -o area area.o -L. -lformas

```

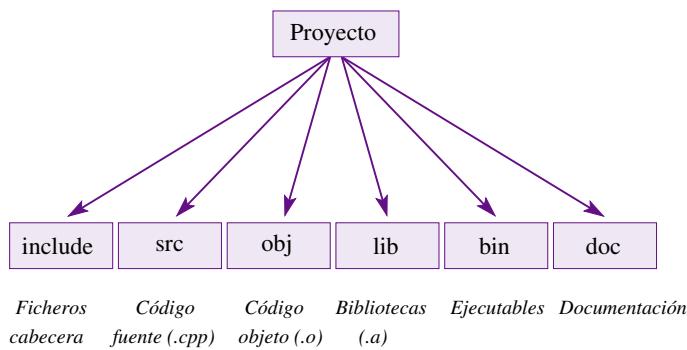
donde vemos que también se presentan las reglas para `central` y `area`, que son los dos programas ejecutables que finalmente queremos generar. Se ha puesto un objetivo —`all`— que depende de `central` y `area` como primer objetivo. La ejecución de `make` sin parámetros lanza la comprobación de la primera regla, por lo que es necesario comprobar las dependencias, lo que provoca la regeneración de los dos ejecutables. Al terminar de comprobar las dependencias, el programa `make` termina al no encontrar acciones que realizar.

Observe que hemos llamado al objetivo `all`, aunque podría haber sido cualquier otro. Este nombre suele ser habitual, ya que expresa la intención de esta regla, es decir, que se ha creado para poder generar *todo*.

**Ejercicio 2.9 — Añadir una aplicación.** Cree una nueva aplicación en un archivo `area.cpp` en la que se pida un círculo al usuario y escriba el área correspondiente. Añada las reglas necesarias al archivo `makefile` para que se generen todos los programas escribiendo `make`.

### Distribución en carpetas

No se tienen que incluir todos los archivos del proyecto en un mismo directorio, sino que se pueden dividir en distintos directorios, separando fuentes de archivos generados y facilitando así el manejo de toda la información. Por ejemplo, se pueden dividir los archivos según el esquema que se presenta en la figura 2.2.



**Figura 2.2**  
Posibles directorios en un proyecto.

En este esquema, el archivo `Makefile` estaría en el directorio `proyecto`; los demás archivos estarían distribuidos en el directorio correspondiente.

Con esta distribución ya no serían válidas las reglas que hemos escrito para gestionar el proyecto, ya que buscaría los ficheros en el directorio actual. Para resolverlo, sería necesario añadir a cada fichero el directorio donde se encuentra. Por ejemplo, la siguiente regla sí tendría en cuenta los directorios donde se encuentran los archivos:

```

1 obj/punto.o : includes/punto.h src/punto.cpp
2     $(CXX) -c $(CXXFLAGS) -o obj/punto.o src/punto.cpp

```

Además, si queremos que los directorios donde se encuentran los archivos puedan cambiarse fácilmente, podemos parametrizar el nombre de estos directorios en un grupo de macros al principio del archivo.

**Ejercicio 2.10 — Distribuir en directorios.** Añada macros `INC`, `SRC`, `OBJ`, `LIB` y `BIN` para parametrizar los directorios donde se encuentran los archivos. Modifique las reglas para tenerlas en cuenta.

Observe que la simple sustitución no es suficiente para que las órdenes de compilación funcionen correctamente. Este problema se debe a que los `#include` de nuestros archivos han dejado de funcionar.

En la situación anterior, el compilador encontraba los archivos que se incluían porque se encontraban en el mismo directorio, pero ahora los archivos `.cpp` no están en el mismo lugar que los `.h`. Para resolverlo, añada la opción `-I` al compilador. Esta opción va seguida de un directorio —similar a la opción `-L` para bibliotecas— donde encontrar archivos cabecera. Bastará con añadirla a la lista de opciones de `CXXFLAGS`, indicando la opción `-I` seguida del directorio que también hemos puesto en la macro `INC`, o mejor aún, seguida de una referencia al contenido de esta macro.

### Otras posibles reglas útiles

A veces se hace necesario borrar ficheros que se consideran temporales o ficheros que no se van a necesitar con posterioridad. Por ejemplo, una vez acabado definitivamente el proyecto y generados los ejecutables, no serán necesarios los códigos objeto ni las bibliotecas, por lo que podemos borrarlos.

Para facilitar esta tarea se suelen incorporar algunas reglas que automatizan estas tareas de limpieza. Es frecuente considerar dos niveles de limpieza del proyecto. Un primer nivel que limpia ficheros intermedios, pero deja las aplicaciones finales que hayan sido generadas:

```
1 clean:
2   echo "Limiando ..."
3   rm $(OBJ)/*.o $(LIB)/lib*.a
```

Con esta regla, tras acabar de programar y depurar el proyecto, podemos quedarnos únicamente con el código fuente y los ejecutables. Observe que esta regla no tiene dependencias, por lo que al aplicarla se pasa directamente a ejecutar sus acciones.

Podríamos crear un segundo nivel que, además de limpiar lo mismo que la regla anterior, también limpie los resultados finales del proyecto:

```
1 mrproper: clean
2   rm $(BIN) /central
```

Con esta regla, lo que conseguimos es que primero se lance la regla `clean` —su dependencia— y que a continuación se apliquen sus acciones. De esta forma obtendremos el proyecto en un estado que contiene únicamente fuentes. Esta situación es la ideal para empaquetar el proyecto y llevarlo a otro lugar para generar, es decir, para distribuir el proyecto con el fin de que sea compilado en otros sistemas.

Por ejemplo, en nuestro proyecto de círculo central, podemos ponernos en el nivel de la carpeta “proyecto” y lanzar la orden de limpieza para a continuación escribir:

```
Consola
prompt> tar zcvf central.tgz proyecto
```

para generar un nuevo archivo `central.tgz` que contiene todos los directorios y archivos que componen el proyecto. Llevando este archivo a otro sistema y ejecutando lo siguiente:

```
Consola
prompt> tar zxvf central.tgz
prompt> cd proyecto
prompt> make
```

podemos obtener los ejecutables. La primera orden despliega el contenido de archivos empaquetados y la última compila todo el proyecto. Observe la conveniencia de una regla `all` para todos nuestros proyectos, o la regla `clean` para generar el proyecto y limpiar los archivos temporales.

### Obtener automáticamente las dependencias

Si su proyecto es bastante complejo y le resulta difícil determinar las dependencias que hay entre los módulos, puede usar el compilador de la *GNU* para que liste las dependencias automáticamente. Para ello, deberá usar la opción `-MM`, que indica al compilador que escriba una regla donde liste los archivos de cabecera que se incluyen. La sintaxis puede ser la siguiente:

```
Consola
prompt> g++ -MM -I include src/circulo.cpp
circulo.o: src/circulo.cpp include/circulo.h include/punto.h
```

Observe que la orden genera una línea con la sintaxis de una regla de `make`. Realmente, la línea que hemos ejecutado es una orden que no llama al compilador sino simplemente al precompilador, ya que es éste el que se ocupa de resolver las directivas `#include`. Además, debemos incluir la opción `-I`, pues el precompilador tiene que incluir los correspondientes archivos cabecera y procesarlos, detectando las inclusiones indirectas.

### 2.3.3 Reglas implícitas

En las secciones anteriores hemos indicado explícitamente una serie de reglas que nos permiten generar los archivos ejecutables. La orden `make` puede, además, usar reglas implícitas, es decir, que no están escritas con todo detalle sino que se presuponen.

Como habrá observado, la forma en que se generan algunos ficheros responde a un *patrón*. Por ejemplo, un archivo `.o` se obtiene desde un `.cpp` lanzando el compilador, seguido de las opciones de compilación, el archivo destino y el archivo fuente.

Por tanto, la orden `make` podría tener en cuenta estas *reglas implícitas* en caso de querer obtener un archivo `.o` a partir de un `.cpp` y no disponer de la regla explícita que indique cómo hacerlo. No sólo el programa `make` “contiene” un conjunto de reglas implícitas predefinidas, sino que podríamos escribir nosotras las nuestras para que se adapten a nuestro proyecto.

Las reglas implícitas son una herramienta muy útil, especialmente en proyectos donde hay un número muy alto de objetivos a obtener. Por ejemplo, imagine que tenemos un proyecto con 50 archivos “`cpp`” que compilar. Sería muy tedioso tener que escribir todas las reglas. Es más fácil indicar un patrón y que lo aplique a todos los archivos.

Dado que nuestros proyectos son relativamente simples, no es necesario hacer uso de este tipo de reglas, sino que podemos escribir explícitamente cada una de ellas. A pesar de ello, es importante tener en cuenta su existencia, ya que es posible que escriba un fichero `makefile`, se olvide de poner alguna regla, y al procesarlo obtener un conjunto de órdenes a las que no encuentra sentido, ya que no las ha escrito sino que han sido generadas a partir de las reglas implícitas.

Finalmente, es importante destacar que los nombres de las macros que se han presentado como de uso habitual —`AR`, `CXX`, etc.— afectan directamente a las reglas implícitas, ya que son éstos los nombres predefinidos que se usan en los patrones de las reglas implícitas.

#### Una solución rápida y últimos detalles

No es objetivo de este documento estudiar la forma de crear complejos ficheros `Makefile`. Pero resulta interesante ofrecer una solución sencilla y rápida para compilar sus proyectos sin tener que dedicar mucho tiempo a diseñar un fichero de este tipo. Para ello, aprovechamos la capacidad de la orden `make` cuando genera reglas implícitas.

En el siguiente ejemplo se muestra un posible fichero `Makefile` para el ejemplo que estamos desarrollando en este guión práctico. En él suponemos que todos los archivos los tenemos en el directorio actual. Observe que los nombres de los fuentes sólo aparecen en las líneas 11 y 12.

```

1 .PHONY: clean mrproper all
2
3 AR = ar
4 # La siguiente es porque CC es el que se usa para enlazar
5 CC = g++
6 CXX = g++
7 CXXFLAGS= -Wall -Wextra -pedantic -std=c++03
8 LDFLAGS= -L.
9 LDLIBS= -lformas
10
11 SOURCESMAIN = central.cpp area.cpp
12 SOURCESLIB = punto.cpp circulo.cpp
13 OBJECTS = $(SOURCESMAIN:.cpp=.o) $(SOURCESLIB:.cpp=.o)
14 EXECUTABLE = $(SOURCESMAIN:.cpp=)
15
16 all: libformas.a $(SOURCESMAIN:.cpp=.o) $(EXECUTABLE)
17
18 libformas.a: $(SOURCESLIB:.cpp=.o)
19     $(AR) rvs $@ $^
20
21 clean:
22     -rm $(OBJECTS)
23
24 mrproper: clean
25     -rm $(EXECUTABLE)
```

Este ejemplo permite por un lado disponer de un archivo simple que es fácil de reutilizar para otros proyectos y, por otro lado, conocer algunos detalles que podrían serle útiles. Concretamente, algunos detalles útiles aunque no use reglas implícitas son:

- Usamos una palabra especial `.PHONY` donde listamos algunos objetivos. Esta palabra informa de los objetivos que son “*falsos*”. Normalmente los objetivos son ficheros por lo que `make` siempre intenta encontrarlos en el disco. En caso de encontrarlos comprobará la necesidad de regenerarlos.
- Si usted crea un archivo que se llame `clean` en disco, la orden `make` lo localiza y al no tener dependencias para regenerarlos, dará por resuelto el objetivo sin lanzar la acción. Por tanto, es conveniente indicar los objetivos que no tiene que comprobar ya que son *ficticios*, de forma que la existencia de ficheros o directorios con ese nombre no afecte al funcionamiento de `make`.
- La orden de borrado tiene un guión como prefijo. No es que la orden sea `-rm`, sino que un guión delante de la orden indica a `make` que ignore lo que devuelva ese programa. Recuerde que los programas devuelven un entero indicando si

han tenido éxito. La orden `make` comprueba siempre ese entero. En caso de que indique un error, detiene la secuencia de acciones y da por terminado el proceso.

Si en nuestro ejemplo no hay archivos objeto o ya los ha borrado, el objetivo `clean` lanza un borrado de archivos que no existen por lo que la orden `rm` devuelve un error y `make` termina. Por ejemplo, si lanza `mrproper`, saltará a `clean`, dará un error y no seguirá con el borrado de los ejecutables.

Por otro lado, aunque no es nuestro objetivo estudiar con detalle la creación y uso de reglas implícitas, le resultará interesante entender que:

- Creamos nuevas variables a partir de otras indicando que hay que hacer una sustitución. Con el carácter ‘`:`’ seguido por una cadena de terminación podemos darle la cadena que la sustituye.
- Hemos creado una regla implícita en la línea 18. Si desea entenderla, puede consultar la bibliografía —por ejemplo, el manual de `make`[16]—. En cualquier caso, seguro que la entiende si informalmente decimos que la línea 19 indica: *Lo que tenga AR, seguido por rvs, seguido por el objetivo, seguido por las dependencias.*

**Ejercicio 2.11 — Reglas implícitas.** Cree un archivo `makefile` con el contenido listado, copie los archivos fuente al mismo directorio y pruebe a ejecutar `make`. Observe cómo las líneas de compilación y enlazado se lanzan a pesar de no haberlas escrito.

Finalmente, una opción que resulta especialmente útil cuando se está probando un archivo `makefile` es la opción `-n`. Cuando la orden `make` se lanza incluyendo esta opción, realiza el proceso de comprobación de reglas pero sin lanzar las acciones. Lo interesante es que escribe en la salida estándar las acciones que se ejecutarían.

**Ejercicio 2.12 — Reglas que se lanzarán.** Ejecute `make mrproper` para limpiar los archivos generados. Después lance `make -n`, compruebe las líneas que escribe y finalmente confirme que los archivos no se han generado.

## 2.4 Mejora y ampliación de la biblioteca

Una vez realizadas las tareas anteriores, donde hemos completado los módulos de gestión de puntos y círculos, así como algún ejemplo de programa para comprobar el correcto funcionamiento, se propone desarrollar dos programas ejecutables que resuelven sendos problemas de procesamiento de puntos. Para ello, se propone la modificación y mejora del software desarrollado. En concreto, se propone:

- La modificación de los módulos para puntos y círculos.
- La ampliación con un nuevo módulo para rectángulos.

Las condición para realizar esta labor de mantenimiento del software es que debemos garantizar que los programas realizados antes de esta modificación deben seguir siendo válidos. Por tanto, no podemos modificar la interfaz si dicha modificación afecta a programas o módulos ya desarrollados.

### 2.4.1 Programas a desarrollar

El alumno debe implementar dos programas nuevos e incorporarlos en las reglas del archivo `Makefile` a fin de que se generen —además de los programas anteriores— los dos nuevos ejecutables. Los programas son:

1. *Longitud de un trayecto.* Calcula la longitud total de una trayecto determinado por una secuencia de 2 o más puntos consecutivos. La longitud será la suma de todos los segmentos rectilíneos que la componen.
2. *Rectángulo delimitador.* Calcula el rectángulo que delimita la región mínima donde se sitúan una secuencia de 1 o más puntos. Un rectángulo vendrá dado por la localización de dos puntos: esquina inferior izquierda y esquina superior derecha.

Dos ejemplos de ejecución para ambos programas se presentan a continuación:

```
prompt> longitud
(0,0) (1,1) (5,1) (5,5)
9.41421
prompt> delimitar
(0,0) (1,1) (5,1) (5,5)
(0,0)-(5,5)
```

Observe que la primera línea corresponde a los datos que introduce el usuario, mientras que la segunda es el resultado escrito por el programa. Si revisa los puntos introducidos, podrá confirmar que tanto la longitud del trayecto como el rectángulo delimitador son correctos.

Por otro lado, tenga en cuenta que podríamos hacer que los puntos se almacenaran en un archivo —por ejemplo, con extensión `pts`— para realizar la misma operación pero asignando el archivo como fuente de entrada estándar. La ejecución sería la siguiente:

```

prompt> longitud < ejemplo.pts
9.41421
prompt> delimitar < ejemplo.pts
(0,0)-(5,5)

```

donde puede ver que no hemos introducido nada por el teclado porque el flujo `cin` se ha asociado al archivo `ejemplo.pts`.

De estos dos ejemplos, puede deducir que los datos se terminan cuando no hay más que leer en la entrada estándar. Es decir, cuando la siguiente lectura de un objeto de tipo *Punto* falla y el flujo queda en estado de fin de archivo (*EOF*).

Finalmente, los programas deberán ser capaces de realizar el mismo cálculo si en lugar de darle los archivos asociándolos a la entrada estándar, damos los nombres de los archivos como parámetros al programa. La ejecución podría ser la siguiente:

```

prompt> longitud ejemplo.pts
9.41421
prompt> delimitar ejemplo.pts
(0,0)-(5,5)

```

## Modificación de los módulos para puntos y círculos

La modificación de una biblioteca implica dos tareas distintas pero igualmente importantes:

1. *Mejorar la biblioteca.* Para ello, podemos plantear la mejora tanto de la interfaz como de la implementación interna. El objetivo es obtener una interfaz más eficaz y un código más eficiente en espacio y/o tiempo.
2. *Mantener la compatibilidad.* Si la biblioteca se ha utilizado antes<sup>3</sup>, deberíamos obtener una nueva versión actualizada que permitiera seguir compilando los programas anteriores.

Si revisa la biblioteca que se ha propuesto, es probable que esté tentado a realizar múltiples cambios. Es más, cuando acabe el curso de C++, seguramente la reescribiría totalmente. Para nuestra práctica, en la que sólo queremos ilustrar la labor de mantenimiento, nos limitaremos a realizar unos pocos cambios.

La actualización de la biblioteca *formas* implica la revisión y mejora de los módulos que gestionan el tipo *Punto* y el tipo *Circulo*. Las modificaciones que deberá realizar en esta práctica son:

- Paso de parámetros por referencia. El tamaño de los objetos es suficientemente grande como para optar por pasarlo por referencia constante. Recuerde que una referencia constante garantiza que no se copiarán los objetos iniciales, sino que se usarán los originales. Para objetos suficientemente grandes podemos evitar la copia del paso por valor si los pasamos por referencia y le añadimos `const` para garantizar que se usarán pero no se modificarán.
- Ampliación de las funciones de E/S. Se incluirá el control de errores en la lectura de datos y la posibilidad de que la fuente/destino de los datos no sean la entrada/salida estándar.

La primera modificación se puede realizar sin problemas, ya que no afecta a la interfaz y los programas anteriores podrán compilarse sin errores.

La segunda es más complicada. En este caso queremos implementar una estrategia de lectura totalmente distinta a la anterior. Una forma sencilla de realizarla es implementar una nueva lectura manteniendo también la anterior. Esta estrategia nos permite seguir manteniendo la compatibilidad aunque nos condene a mantener las funciones anteriores.

La cabecera de la nueva función podría ser la siguiente:

```
bool Leer(istream& is, Punto& p);
```

que recibe un flujo de entrada y un punto que leer y nos devuelve si ha tenido éxito.

Esta función es válida para leer un punto en formato de texto desde cualquier flujo de entrada. Podemos pasar el objeto `cin` como primer parámetro o un objeto de tipo `ifstream` asociado a un archivo de disco.

Lógicamente, nuestros programas seguirán siendo válidos porque mantenemos las funciones de E/S anteriores, a pesar de que ésta es una opción más interesante. Una solución intermedia a mantener o no las anteriores es mantenerla pero marcarlas como "obsoletas"<sup>4</sup>. La idea es que se sigue manteniendo para que el software siga siendo compatible pero se avisa de que en las siguientes versiones de la biblioteca podría desaparecer.

Las nuevas funciones de E/S que deberá incluir en los módulos anteriores son las siguientes:

```
bool Leer(istream& is, Punto& p);
bool Leer(istream& is, Circulo& c);
bool Escribir(ostream& os, const Punto& p);
bool Escribir(ostream& os, const Circulo& c);
```

Observe que se repiten los nombres, aunque el compilador no tendrá ningún problema en distinguirlas ya que conoce el tipo de los parámetros.

<sup>3</sup>En nuestro caso podríamos modificarla y, si es necesario, cambiar los programas anteriores. Sin embargo, suponga que se han escrito múltiples programas también por otros usuarios y queremos ofrecerles una mejor biblioteca sin que tengan que reescribir su código.

<sup>4</sup>Del inglés "deprecated".

Por otro lado, es posible que necesite alguna función adicional para crear el programa concreto. Son funciones que facilitan la solución del programa aunque no está claro que se vayan a reutilizar en el futuro. Se pueden incluir en el archivo `cpp` que contiene el `main`. Por ejemplo, puede crear:

```
double Longitud(istream& is);
Rectangulo BoundingBox(istream& is);
```

para cada uno de los programas a realizar, respectivamente. Es interesante que ya en este ejemplo reflexione sobre la posibilidad de añadirlas como funciones de la biblioteca. Si no tenemos una idea clara, mejor es ser conservadores y no hacerlo.

Recuerde que si no las incluimos y en el futuro queremos incluirlas será fácil moverlas. Sin embargo, si las incluimos, tendremos que mantenerlas en la biblioteca aunque descubramos que fue un error; recuerde los comentarios sobre la eliminación o no de las funciones de lectura anteriores.

## Módulo para rectángulos

Creamos un nuevo tipo `Rectangulo` para almacenar un rectángulo como una estructura que necesita dos miembros: un punto que indica la esquina inferior izquierda y un punto que indica la esquina superior derecha. Con esta estructura podemos asociar una serie de operaciones:

```
bool Leer(istream& is, Rectangulo& r);
bool Escribir(ostream& os, const Rectangulo& r);
void InicializarRectangulo (Rectangulo& r, const Punto& p1, const Punto& p2);
Punto InferiorIzquierda (const Rectangulo& r);
Punto SuperiorDerecha (const Rectangulo& r);
double Area(const Rectangulo& r);
bool Interior (const Punto& p, const Rectangulo& r)
```

que deberán incluirse como un nuevo módulo de la biblioteca `formas`. Para implementar estas funciones tenga en cuenta que:

- Los puntos para la función `IniciarRectangulo` pueden estar en cualquier orden o incluso corresponder al punto superior izquierda e inferior derecha.
- El formato de lectura y escritura de puntos son dos puntos con un guion entre ellos.
- No se han documentado, pero el resto de funciones tienen un significado fácilmente deducible desde sus nombres.

## Formato de archivo de puntos

Una vez compruebe que el programa le funciona correctamente, deberá modificar las funciones de lectura para poder trabajar con archivos de datos que contienen comentarios.

Un archivo de puntos contendrá una secuencia de puntos en el formato indicado —paréntesis, `x`, coma, `y`, paréntesis— que estarán separados por “espacios blancos”<sup>5</sup>, es decir, caracteres espacio, tabulador, salto de línea, etc.

Además, el archivo puede contener comentarios internos para documentar y editar más cómodamente el contenido de un archivo. Estos comentarios comienzan en el carácter `#` y llegan hasta el final de la línea. La lectura de un dato desde un archivo consiste en: primero descartar todos los “espacios blancos” y comentarios hasta el siguiente punto, y segundo leer el punto correspondiente. Un ejemplo de este archivo es el fichero `ejemplo.pts` que puede encontrar junto a esta práctica.

## Parámetros de la función `main`

Finalmente, dado que probablemente no habrá trabajado con parámetros en la línea de órdenes, es interesante incluir una breve reseña sobre la forma en que puede manejarlos.

Cuando queremos manejar los parámetros de línea de órdenes optamos por otra cabecera de la función `main`. Concretamente, la función sería la siguiente:

```
int main(int argc, char* argv[])
{
    // Cuerpo de la función
}
```

El comportamiento de la función es idéntico al que ya conoce, aunque ahora tenemos dos parámetros —`argc`, `argv`— disponibles. Estos parámetros indican cada uno de los valores de las cadenas de caracteres que se han dado en la llamada al programa. Concretamente, los parámetros nos permiten acceder a:

- `argv[0]`: cadena que contiene el nombre del programa.
- `argv[1]`: cadena que corresponde al primer parámetro después del nombre del programa.
- `argv[2]`: cadena que corresponde al segundo parámetro.
- ...
- `argv[argc-1]`: último parámetro de la línea de órdenes.

Por tanto, una llamada a nuestro programa recibirá un valor de `argc` de uno cuando no hay argumentos en la línea de órdenes y un valor de dos cuando le hemos dado el nombre del archivo. Este nombre de archivo estará almacenado en `argv[1]`.

<sup>5</sup>Consulte el manual de la función `isspace` de `cstring`.

## Código de ejemplo

Para este guión se incluye un programa completo que resuelve un problema similar a los propuestos. Es probable que el estudiante aún tenga muy poca experiencia y aunque tenga los conocimientos para proponer una solución, le resulte difícil obtener, al menos, una buena solución.

En esta sección incluimos un programa con un diseño similar al que se ha pedido. Tenga en cuenta que deberá estudiar este código, entender por qué funciona, y usar esas conclusiones para proponer la solución que necesita para esta práctica. Como puede ver, esta sección corresponde a un ejercicio de lectura de “código de terceros”. La idea es aprender con ejemplos, ya que la lectura de código de programadores con más experiencia resulta una fuente muy importante de aprendizaje, en su primer curso y el resto de su vida profesional como programador.

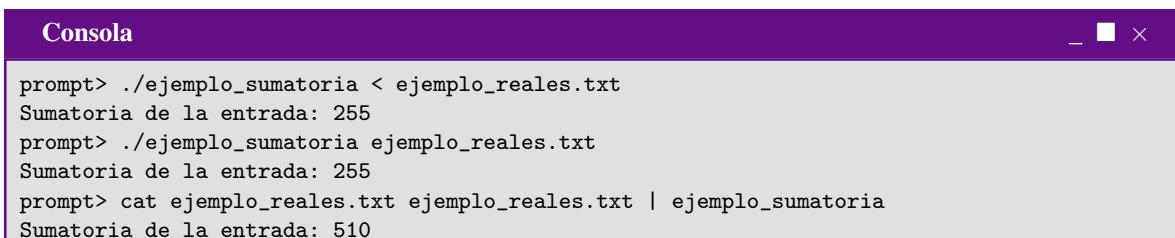
En el siguiente listado se presenta un programa que resuelve la sumatoria de una serie de número que se dan por la entrada estándar o con un nombre de archivo. Es decir, tiene una interfaz similar a la propuesta para los programas `longitud` y `rectangulo`:

### Listado 1 — Sumatoria de números.

```

1 #include <fstream>
2 #include <iostream>
3 using namespace std;
4
5 void Avanzar(std::istream& is)
6 {
7     while (isspace(is.peek()) || is.peek()=='#') {
8         if (is.peek()=='#')
9             is.ignore(1024, '\n'); // Suponemos una línea tiene menos de 1024
10        else is.ignore();
11    }
12 }
13
14 double Sumatoria(std::istream& is)
15 {
16     double s=0, dato;
17
18     Avanzar(is);
19     while (is>> dato) { // Mientras tenga éxito la lectura
20         s+=dato;
21         Avanzar(is); // Descarta comentarios y para en siguiente dato
22     }
23     return s;
24 }
25
26 int main(int argc, char* argv[])
27 {
28     double sumatoria= 0;
29     bool fin_entrada;
30     if (argc==1) { // Si no hemos dado parámetros en la línea de órdenes
31         sumatoria= Sumatoria(cin);
32         fin_entrada=cin.eof();
33     }
34     else {
35         ifstream f(argv[1]); // Como parámetro, el nombre del archivo
36         if (!f) {
37             cerr << "Error: no se abre " << argv[1] << endl;
38             return 1;
39         }
40         sumatoria=Sumatoria(f);
41         fin_entrada=f.eof();
42     }
43
44     if (!fin_entrada) {
45         cerr << "Error inesperado. No se ha leído toda la entrada" << endl;
46         return 1;
47     }
48     cout << "Sumatoria de la entrada: " << sumatoria << endl;
49 }
```

Puede usar el programa para sumar una serie de número en un archivo que contiene datos mezclados con comentarios delimitados por un carácter '#'. En el paquete que se ofrece junto con este guón encontrará el anterior listado junto con un archivo de ejemplo para poder probarlo. Algunas ejecuciones son:



```

prompt> ./ejemplo_sumatoria < ejemplo_reales.txt
Sumatoria de la entrada: 255
prompt> ./ejemplo_sumatoria ejemplo_reales.txt
Sumatoria de la entrada: 255
prompt> cat ejemplo_reales.txt ejemplo_reales.txt | ejemplo_sumatoria
Sumatoria de la entrada: 510

```

Observe que en la primera está leyendo los datos desde `cin` y en la segunda tiene que abrir el archivo para realizar la misma operación. En la tercera ejecución hemos creado un flujo que encadena dos veces el mismo archivo para que entre por la entrada estándar al programa. Descargue el programa y el ejemplo para estudiar los detalles antes de abordar la última parte de la práctica.

## 2.4.2 Práctica a entregar

El alumno deberá empaquetar todos los archivos relacionados en el proyecto en un archivo con nombre `makes.tgz` y entregarlo en la fecha que se publicará en la página web de la asignatura.

Tenga en cuenta que no se incluirán ficheros objeto ni ejecutables. Es recomendable que haga una “*limpieza*” para eliminar los archivos temporales o que se pueden generar a partir de los fuentes. Para realizar la entrega, en primer lugar, realice la limpieza de archivos que no se incluirán en ella. Una vez que ha eliminado los archivos generados y sólo tiene los fuentes, sitúese en la carpeta superior para ejecutar:



```
Consola
prompt> tar zcvf makes.tgz proyecto
```

tras lo cual, dispondrá de un nuevo archivo `makes.tgz` que contiene la carpeta `proyecto`, así como todas las carpetas y archivos que cuelgan de ella. Recuerde que esta carpeta se encontrarán todas las carpetas con los archivos distribuidos. El archivo `makefile` que entregará contendrá todas y cada una de las reglas necesarias, incluyendo los caminos relativos a la localización en subdirectorios.

# 3

## Esteganografía: texto

Introducción.....	27
Tipo enumerado	
Operadores a nivel de bit	
Imagenes .....	29
Niveles de gris y color	
Funciones de E/S de imágenes	
Ocultar/Revelar un mensaje.....	32
Desarrollo de la práctica .....	32
Módulo codificar	
Programas	
Práctica a entregar	

### 3.1 Introducción

Los objetivos de este guión de prácticas son los siguientes:

1. Usar tipos de datos **enum**.
2. Practicar con operaciones a nivel de bit para acceder a la representación de tipos integrales.
3. Practicar con *vectores-C*, incluyendo el caso particular de *cadenas-C*.
4. Incorporar código de terceros en el programa.
5. Practicar el diseño de programas con módulos independientes así como la compilación separada, incluyendo la creación de bibliotecas.
6. Introducirse en el uso de **doxygen** como forma de documentación de código.
7. Crear y usar archivos **makefile** para gestionar el proyecto.

Por otro lado, también se incluye una introducción simple a las imágenes. Este conocimiento se podría considerar a nivel de usuario, por ejemplo, de un programa de procesamiento de imágenes. Con ello, también podemos considerar como objetivos:

1. Conocer las imágenes como una matriz de objetos que especifican un nivel de gris o color.
2. Conocer el espacio de color *RGB* y cómo se pueden manejar tripletas de valores para generar una amplia gama de colores.

El alumno debe realizar esta práctica una vez que haya estudiado los contenidos indicados en los puntos anteriores. No será necesario —de hecho no se permite— usar tipos puntero<sup>1</sup> o memoria dinámica, e incluso los tipos de la STL como **vector** o **string**.

En esta sección introductoria se incluye un repaso breve sobre el tipo enumerado y los operadores a nivel de bit para hacer la práctica más autocontenido. Tenga en cuenta que además del tipo enumerado, serán necesarios:

1. Conocimientos sobre *vectores-C*, incluyendo los vectores de **char** para almacenar una *cadena-C*.
2. Conocimientos sobre compilación separada y gestión de proyectos con **make**.

Después de realizar esta práctica, debería ser capaz de entender no sólo el funcionamiento de los tipos básicos y los vectores, sino también las limitaciones y necesidades de éstos. El resultado debería motivar el estudio de otros temas más avanzados.

#### 3.1.1 Tipo enumerado

En muchos casos es necesario manejar un tipo de dato que pueda tener un conjunto finito y pequeño de posibles valores. Algunos ejemplos son:

- Día de la semana: con 7 posibles valores, de lunes a domingo.
- Mes del año: con 12 valores desde enero a diciembre.
- Palos de la baraja española: con valores oros, copas, espadas y bastos.

Para estos casos, una solución directa y muy sencilla es el uso de un tipo de dato entero predefinido. Podemos usar, por ejemplo, los valores de 0 a 6 para indicar un día de la semana, de 1 a 12 para un mes, o de 1 a 4 para los palos de la baraja.

Sin embargo, resulta mucho más legible usar un tipo de dato que refleje mejor el tipo de objeto que se maneja, así como sus posibles valores. Para ello, el lenguaje nos permite crear nuevos tipos de datos enumerados —pues se crean indicando todos y cada uno de sus valores— con la palabra reservada **enum**. La sintaxis para declarar el nuevo tipo es:

<sup>1</sup>En realidad, el programa contiene tipos puntero, pero el programador no necesita saber de su existencia para obtener la solución.

```
enum <nombre_del_tipo> { lista de posibles valores }
```

Por ejemplo, se podrían crear dos tipos de datos DiaSemana y Mes, de forma que sea más legible el código que maneja este tipo de objetos. Por ejemplo, podemos obtener el siguiente código:

```
// Un tipo de dato para manejar el día de la semana
enum DiaSemana {LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO, DOMINGO};

// Un tipo de dato para manejar el mes
enum Mes {ENERO, FEBRERO, MARZO, ABRIL, MAYO, JUNIO,
JULIO, AGOSTO, SEPTIEMBRE, OCTUBRE, NOVIEMBRE, DICIEMBRE};

// El segundo parámetro es de tipo Mes
// Devuelve un objeto de tipo DiaSemana
DiaSemana ObtenerDiaSemana(int d, Mes m, int a) {
    // ... código...
}

DiaSemana cae_en;

cae_en= ObtenerDiaSemana(25,DICIEMBRE,2011);
if (cae_en==DOMINGO)
    cout << "La navidad cae en domingo..." << endl;
```

La forma en que el compilador maneja estos nuevos tipos internamente es muy simple, ya que prácticamente lo que hace es considerar que los nuevos tipos almacenan algún tipo de entero y que cada uno de los valores concretos que se enumeran representan una constante entera. La intención de este comentario no es que entienda cómo funciona internamente el compilador. La intención es que no se sorprenda cuando al realizar una operación extraña descubra que el compilador “no se queja” y termina obteniendo un resultado inesperado. Por ejemplo, si multiplica un valor de mes por 3, verá que el código se compila.

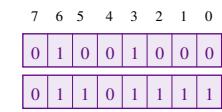
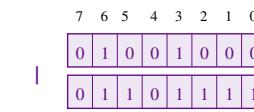
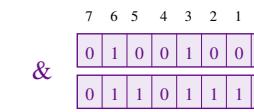
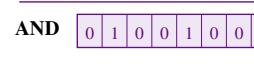
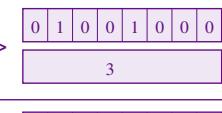
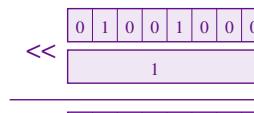
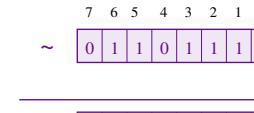
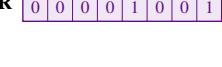
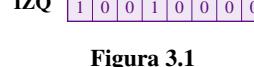
En esta práctica nos limitaremos, fundamentalmente, a usar los valores concretos de un enumerado y a realizar comparaciones como las presentadas en el ejemplo anterior.

### 3.1.2 Operadores a nivel de bit

El lenguaje C++ ofrece un conjunto de operadores lógicos a nivel de bit para operar con tipos integrales y enumerados, en particular, con tipos carácter y entero. Los operadores son:

- *Operador unario*. La operación se realiza sobre todos y cada uno de los bits que contiene el operando.
  - No ( $\sim exp1$ ). Obtiene un nuevo valor con los bits cambiados, es decir, ceros por unos y unos por ceros.
- *Operadores binarios*. La operación se realiza para cada par de bits que ocupan igual posición en ambos operandos.
  - O exclusivo ( $exp1 \wedge exp2$ ) bit a bit. Es decir, obtiene 1 cuando uno, y sólo uno de los operandos, vale 1.
  - O ( $exp1 | exp2$ ). Obtiene 1 cuando alguno de los dos operandos vale 1.
  - Y ( $exp1 \& exp2$ ). Obtiene 1 sólo si los dos operandos valen 1.
- *Operadores de desplazamiento*. Se desplazan los bits a derecha o izquierda de forma que algunos bits se pierden. Será necesario insertar nuevos bits, que tendrán valor cero.
  - Desplazamiento a la derecha ( $exp1 >> exp2$ ). Los bits del primer operando se desplazan a la derecha tantos lugares como indique el segundo operando.
  - Desplazamiento a la izquierda ( $exp1 << exp2$ ). Los bits del primer operando se desplazan a la izquierda tantos lugares como indique el segundo operando.

En la figura 3.1 se muestran algunos ejemplos con los operadores que hemos indicado.

$\wedge$		$ $		$\&$	
<b>XOR</b>		<b>OR</b>		<b>AND</b>	
$>>$		$<<$		$\sim$	
<b>DER</b>		<b>IZQ</b>		<b>NOT</b>	

**Figura 3.1**  
Ejemplos de operadores a nivel de bit.

Con estos operadores, se pueden crear expresiones para consultar el valor de los bits que componen un dato, o para modificarlos. Veamos cómo se podría resolver.

### Consulta del valor de un bit

Para consultar si un bit vale cero o uno podemos realizar una operación *AND*. Por ejemplo, si queremos consultar el valor del bit 3, creamos un dato que tiene todo cero excepto ese bit y realizamos una operación de *AND* bit a bit. Con este operador, cualquier bit que se opere con el cero obtendrá el valor cero, mientras que si se opera con el uno obtendrá el valor del bit. En la figura 3.2 se muestra esta idea gráficamente.

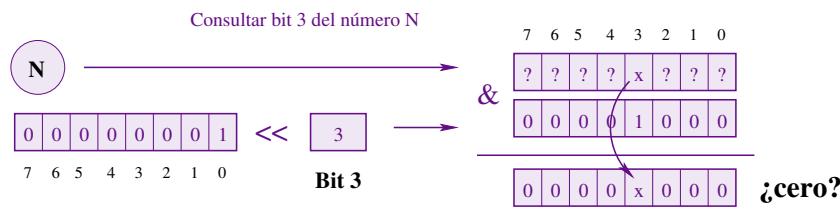


Figura 3.2  
Consulta del valor de un bit.

Observe que el valor que resulta en la posición 3 es exactamente el mismo que tenía el dato inicial *N*. Si el bit valía cero, el resultado de la operación es una tupla con todos ceros, es decir, el número cero. Si valía 1 se obtiene una tupla que tiene un bit activado, es decir, un valor distinto de cero.

### Modificar el valor de un bit

La operación de modificar un bit depende de si queremos activar o desactivar el bit. En la figura 3.3 se presenta gráficamente cada una de las dos operaciones.

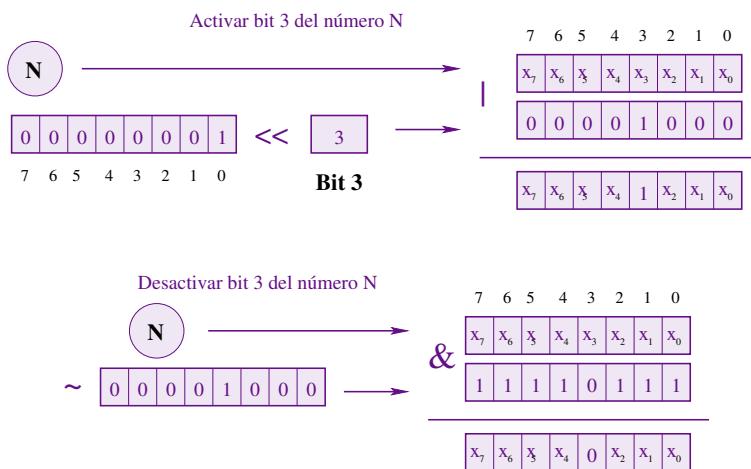


Figura 3.3  
Modificación del valor de un bit.

Observe que la activación —hacer 1 el bit— consiste en una operación *OR* con un dato que tiene un bit 1 en la posición a activar. En cambio, la operación para desactivar —hacer 0 el bit— consiste en hacer un *AND* con un dato que tiene un único cero en la posición a desactivar. Como puede ver, ese dato se obtiene con una operación *NOT* sobre el anterior.

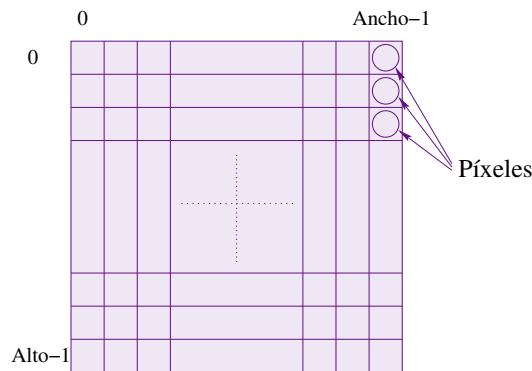
Finalmente, tal vez le interese modificar el valor de un bit de forma que si vale cero se convierta en uno y al revés. En este caso es tentador comprobar el valor que tiene inicialmente y ejecutar la operación correspondiente. Sin embargo, en la práctica es mucho más sencillo, pues basta hacer una operación *XOR* con un dato que tiene un único bit 1 en la posición a cambiar. Recuerde que *bit XOR 1 = NOT bit*.

## 3.2 Imágenes

Desde un punto de vista práctico, una imagen se puede considerar como un conjunto de celdas o píxeles que se organizan en posiciones que podemos hacer corresponder con una matriz —bidimensional— tal como muestra la figura 3.4.

El contenido de cada una de las celdas dependerá en gran medida de la aplicación donde se quiera utilizar. Algunos ejemplos podrían ser:

- Una imagen que haga de máscara para situar los puntos donde se encuentra cierta información relevante de otra imagen, es decir, una imagen para la que queremos guardar una información binaria para cada punto. En este caso, bastaría con almacenar un bit en cada una de las celdas.
- Una imagen de luz. Si queremos almacenar una escena en blanco y negro, podemos crear un rango de valores de luminosidad (que llamaremos a partir de ahora valores de gris), por ejemplo los enteros en el rango [0,255] (el cero es negro, y el 255 blanco). En este caso, cada celda puede almacenar un único byte.



**Figura 3.4**  
Imagen como estructura bidimensional de píxeles.

- Si queremos almacenar una imagen médica, donde cada punto mantiene la densidad obtenida a partir de un aparato de rayos X, el rango de posibles valores podría estar en  $[0, 4096]$ . Cada celda almacenaría un valor de este rango, por ejemplo, un entero corto.
- Si queremos almacenar una escena con información de color, podemos fijar en cada celda una tripleta de valores indicando el nivel de intensidad con el que contribuyen 3 colores básicos para formar el color requerido.

En la práctica que se propone en este documento, trabajaremos con imágenes de grises y en color.

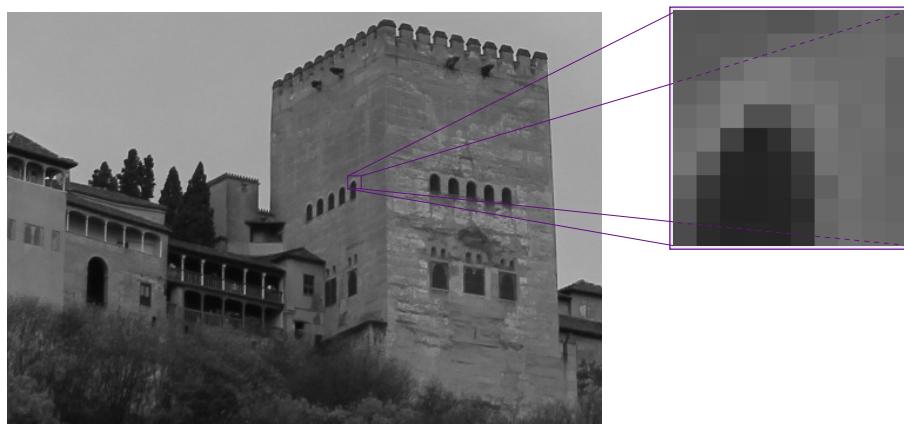
### 3.2.1 Niveles de gris y color

En esta práctica vamos a considerar dos tipos de imágenes: en niveles de gris y en color. La primera de ellas corresponde a una imagen en blanco y negro, mientras que la segunda podrá tener cualquier color de una gama de  $256^3 = 16777216$  colores.

Para representar las imágenes en blanco y negro podemos usar un rango de valores para indicar todas las tonalidades de gris que van desde el negro hasta el blanco. En nuestro caso, las imágenes almacenarán en cada píxel un valor de gris desde el 0 al 255. Por ejemplo, un píxel con valor 128 tendrá un gris intermedio entre blanco y negro.

La elección del rango  $[0, 255]$  se debe a que esos valores son los que se pueden representar en un byte<sup>2</sup>(8 bits). Por tanto, si queremos almacenar una imagen de niveles de gris, necesitaremos  $\text{ancho} \cdot \text{alto}$  bytes. En una imagen de 256 por 256 píxeles, necesitaríamos 64 Kbytes para representar todos sus píxeles.

En la figura 3.5 se muestra un ejemplo de imagen 500x350 de niveles de gris. Observe el zoom de una región 10x10 para apreciar con detalle los grises que la componen.



**Figura 3.5**  
Imagen de niveles de gris.

Para representar un color de forma numérica, no es posible usar un único valor, sino que se deben incluir tres números. Existen múltiples propuestas sobre el rango de valores y el significado de cada una de esas componentes, generalmente adaptadas a diferentes objetivos y necesidades.

En una imagen en color, el contenido de cada píxel será una tripleta de valores según un determinado modelo de color. En esta práctica consideraremos el modelo RGB. Este modelo es muy conocido, ya que se usa en dispositivos como los monitores, donde cada color se representa como la suma de tres componentes: rojo, verde y azul<sup>3</sup>.

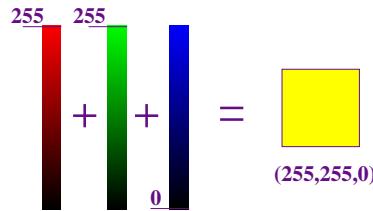
Podemos considerar distintas alternativas para el rango de posibles valores de cada componente, aunque en la práctica, es habitual asignarle el rango de números enteros desde el 0 al 255, ya que permite representar cada componente con un único

<sup>2</sup>Recuerde que en nuestro caso, el tipo más adecuado para almacenar este rango es `unsigned char`.

<sup>3</sup>Red, Green, Blue.

byte, y la variedad de posibles colores es suficientemente amplia. Por ejemplo, el ojo humano no es capaz de distinguir un cambio de una unidad en cualquiera de las componentes.

En la figura 3.6 se muestra un ejemplo en el que se crea un color con los valores máximos de rojo y verde, con aportación nula del azul. El resultado es el color (255,255,0), que corresponde al amarillo.



**Figura 3.6**  
Color como mezcla RGB.

### 3.2.2 Funciones de E/S de imágenes

No es intención en la práctica introducir demasiados detalles sobre imágenes. Para resolver el problema de la E/S introduciremos un formato muy simple, sin compresión y sin pérdida de información que puede servir como introducción y que permite al estudiante ser capaz de entender el código que resuelve el problema. Concretamente, las imágenes que manearemos están almacenadas en un fichero que se divide en dos partes:

1. *Cabecera.* En esta parte se incluye información acerca de la imagen, sin incluir el valor de ningún píxel concreto. Así, podemos encontrar valores que indican el tipo de imagen que es, comentarios sobre la imagen, el rango de posibles valores de cada píxel, etc. En esta práctica, esta parte nos va a permitir consultar el tipo de imagen y sus dimensiones sin necesidad de leerla.
2. *Información.* Contiene los valores que corresponden a cada píxel. Hay muchas formas para guardarlos, dependiendo del tipo de imagen de que se trate, pero en nuestro caso será muy simple, ya que se guardan todos los bytes por filas, desde la esquina superior izquierda a la esquina inferior derecha.

Los tipos de imagen que vamos a manejar serán **PGM** (*Portable Grey Map file format*) y **PPM** (*Portable Pix Map file format*), que tienen un esquema de almacenamiento con cabecera seguida de la información, como hemos indicado. El primero se usará para las imágenes en blanco y negro y el segundo para las imágenes en color.

Para simplificar la E/S de imágenes de disco, se facilita un módulo —archivo de cabecera y de definiciones— que contiene el código que se encarga de resolver la lectura y escritura de ambos formatos. Por tanto, el alumno no necesitará estudiar los detalles de cómo es el formato interno de estos archivos. En lugar de eso, deberá usar las funciones proporcionadas para resolver ese problema. El archivo de cabecera contiene lo siguiente:

```
#ifndef _IMAGEN_ES_H_
#define _IMAGEN_ES_H_

enum TipoImagen { IMG_DESCONOCIDO, ///< Tipo de imagen desconocido
                  IMG_PGM,           ///< Imagen tipo PGM
                  IMG_PPM            ///< Imagen tipo PPM
};

// Devuelve el tipo. IMG_DESCONOCIDO si el fichero no existe o no es compatible
TipoImagen LeerTipoImagen (const char nombre[],           // nombre del archivo a localizar
                           int& filas,                 // filas de la imagen si es PGM/PPM
                           int& columnas);           // columnas de la imagen si es PGM/PPM

// Devuelve si ha tenido éxito la lectura de una imagen PGM
bool LeerImagenPGM (const char nombre[],                // nombre del archivo a leer
                     int& filas,                  // filas de la imagen leída
                     int& columnas,              // columnas de la imagen leída
                     unsigned char buffer[]);    // lugar donde meter los valores

// Devuelve si ha tenido éxito la escritura de una imagen PGM
bool EscribirImagenPGM (const char nombre[],           // nombre del archivo a crear
                        const unsigned char datos[], // valores de los píxeles
                        int filas,                  // filas de la imagen a crear
                        int columnas);             // columnas de la imagen a crear

// Devuelve si ha tenido éxito la lectura de una imagen PPM
bool LeerImagenPPM (const char nombre[],           // nombre del archivo a leer
                     int& filas,                  // filas de la imagen leída
                     int& columnas,              // columnas de la imagen leída
                     unsigned char buffer[]);    // donde almacenar los valores RGBRGBRGB...

// Devuelve si ha tenido éxito la escritura de una imagen PPM
bool EscribirImagenPPM (const char nombre[],           // nombre del archivo a crear
                        const unsigned char datos[], // valores de los píxeles RGBRGBRGB...
                        int filas,                  // filas de la imagen a salvar
                        int columnas);             // columnas de la imagen a salvar

#endif
```

Además, se incluye documentación en formato `doxygen` para que sirva de muestra y pueda ser usada como referencia para estas funciones. Ejecute `make documentacion` en el paquete que se adjunta para obtener la salida de esa documentación en formato `HTML` (use un navegador para consultarla).

Si estudia detenidamente las cabeceras de las funciones que se proporcionan, verá que es fácil intuir el objetivo de cada uno de ellas. Tal vez, la parte más confusa pueda surgir en los parámetros correspondientes al `buffer` o los datos de la imagen (vectores de `unsigned char`):

1. Si la imagen es `PGM` —de grises— será un vector que contenga todos los bytes consecutivos de la imagen. La posición 0 del vector tendrá el píxel de la esquina superior izquierda, la posición 1 el de su derecha, etc.
2. Si la imagen es `PPM` —de color— será un vector similar. En este caso, la posición 0 tendrá la componente R de la esquina superior izquierda, la posición 1 tendrá la G, la posición 2 la B, la posición 3 la componente R del siguiente píxel, etc. Es decir, añadiendo las tripletas RGB de cada píxel.

### 3.3 Ocultar/Revelar un mensaje

Este guión práctico sobre esteganografía propone implementar un método muy concreto con el que se inserta o extrae un mensaje “oculto” en una imagen. El método consiste en modificar el valor de cada píxel para que contenga parte de la información a ocultar. Ahora bien, ¿Cómo almacenamos un mensaje (*cadena-C*) dentro de una imagen?

Tenga en cuenta que los valores que se almacenan en cada píxel corresponden a un valor en el rango [0,255] y que, por tanto, el contenido de una imagen no es más que una secuencia de valores consecutivos en este rango. Si consideramos que el ojo humano no es capaz de detectar cambios muy pequeños en dichos valores, podemos insertar el mensaje deseado modificando ligeramente cada uno de ellos. Concretamente, si cambiamos el valor del bit menos significativo<sup>4</sup>, habremos afectado al valor del píxel, como mucho, en una unidad de entre las 255. La imagen la veremos, por tanto, prácticamente igual.

Nuestro programa va a cambiar una imagen, pero sólo los bits menos significativos de cada píxel. Es decir, disponemos del bit menos significativo para cambiarlo como deseemos ya que no vamos a poder distinguirlo visualmente. Será en estos bits menos significativos donde codificaremos el mensaje.

El mensaje será una *cadena-C*, es decir, una secuencia de valores de tipo `char` que terminan en un cero. El mensaje es, por tanto, una secuencia de bytes (8 bits) que queremos insertar en la imagen. Dado que podemos modificar los bits menos significativos de la imagen, podemos “repartir” cada carácter del mensaje en 8 píxeles consecutivos. En la figura 3.7 mostramos un esquema que refleja esta idea.

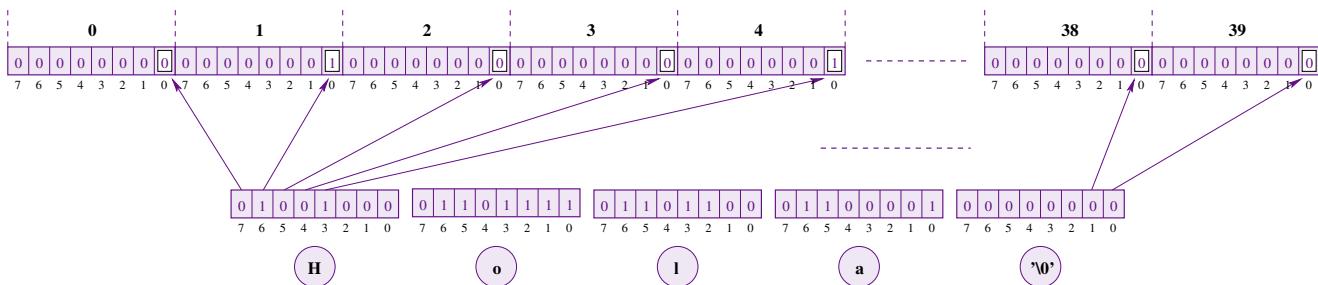


Figura 3.7  
Inserción de un mensaje en el bit menos significativo.

Como puede ver, la secuencia de 40 octetos —bytes— superior corresponde a los valores almacenados en el vector de objetos `unsigned char` que corresponde a la imagen. Podemos suponer, por ejemplo, que la imagen es negra y que por tanto todos los píxeles de la figura tienen un valor cero.

En la fila inferior, podemos ver un mensaje con 4 caracteres —5 incluyendo el cero final— que corresponde a la secuencia a ocultar. Observe que se han repartido en la secuencia superior de forma que la imagen ha quedado modificada, aunque visualmente no podremos distinguir la diferencia.

Para realizar la extracción del mensaje tendremos que resolverlo con la operación inversa, es decir, tendremos que consultar cada uno de esos bits menos significativos y colocarlos de forma consecutiva, creando una secuencia de octetos —bytes— hasta que extraigamos un carácter cero.

Por último, es interesante destacar que en el dibujo hemos representado una distribución de bits de izquierda a derecha. Es decir, el bit más significativo se ha insertado en el primer byte, el siguiente en el segundo, hasta el menos significativo que se ha insertado en el octavo. El estudiante debe realizar la inserción en este orden y tenerlo en cuenta cuando esté revelando el mensaje codificado.

### 3.4 Desarrollo de la práctica

Para resolver este problema, el alumno tendrá que llevar a cabo una serie de tareas —que exponemos en esta sección— junto con las condiciones o restricciones que deberá tener en cuenta.

<sup>4</sup>El que representamos a la derecha, y que corresponde a las unidades del número binario.

Antes de comenzar, el alumno debe descargar un paquete con el material y datos básicos a partir del cual desarrollar la práctica. En este paquete, por ejemplo, encontrará el código que resuelve el problema de la E/S de imágenes, así como alguna imagen de ejemplo.

### 3.4.1 Módulo codificar

La tarea básica que hay que realizar en la práctica consiste en la inserción y extracción de un mensaje en una serie de bytes que pertenecen a una imagen. Por tanto, en primer lugar, se propone la creación de un nuevo módulo “codificar”, que se encargue de esa tarea y que se use para enlazarse con los programas que se van a desarrollar. Este módulo contendrá dos funciones:

- Función *Ocultar*, que recibe como entrada dos parámetros, uno con la imagen (un vector de bytes) y otro con el mensaje a insertar (una *cadena-C*). Esta función insertará el mensaje en la imagen.
- Función *Revelar*. Recibe como parámetros la imagen (un vector) y una cadena (un vector de caracteres), que se modificará para contener el mensaje que se va a extraer desde el vector.

Debe tener en cuenta que se pueden dar situaciones de error y los programas deberán actuar adecuadamente. Ejemplos de posibles situaciones de error:

- La cadena que se intenta codificar es demasiado grande para la imagen dada.
- La imagen que se supone con mensaje oculto no contiene ningún carácter terminador de cadena (carácter '`\0`').
- La imagen oculta un mensaje de tamaño mayor que el parámetro cadena que se le pasa a la función *Revelar*.

Si lo considera oportuno, puede añadir parámetros adicionales a las dos funciones propuestas para tener en cuenta el tamaño de los vectores y cadenas. De esa forma, las mismas funciones podrán procesar las situaciones de error. Por ejemplo, pueden devolver un valor que indique si ha habido algún error.

También puede optar por imponer precondiciones a esas funciones, en cuyo caso, debería comprobar que no hay errores —se cumplen las condiciones— en el lugar de la llamada.

El alumno debe crear los archivos `codificar.h` y `codificar.cpp` para resolver estos dos problemas. Tenga en cuenta que puede incluir la devolución de algún valor que indique si se ha conseguido realizar la operación con éxito.

### Documentación

Si revisa el material que se ha descargado para desarrollar la práctica, encontrará que el módulo de E/S de imágenes está documentado de acuerdo a la sintaxis de *doxygen*. De hecho, incluso se ha proporcionado lo necesario para poder realizar fácilmente la generación de la documentación asociada en formato html.

De igual forma, el alumno debe añadir la documentación del módulo que ha desarrollado. Más concretamente, añadir comentarios *doxygen* al fichero `codificar.h` que ha creado.

### 3.4.2 Programas

El objetivo final de la práctica es crear dos programas, uno para ocultar un mensaje en una imagen y otro para revelarlo.

#### Ocultar

El programa de ocultación debe insertar un mensaje en una imagen. El programa pide en consola el nombre de la imagen de entrada, el nombre de la imagen de salida, y el mensaje a insertar. Un ejemplo de ejecución podría ser el siguiente:

```
Consola
prompt> ocultar
Introduzca la imagen de entrada: original.ppm
Introduzca la imagen de salida: salida
Introduzca el mensaje: ;Hola mundo!
Ocultando...
prompt>
```

donde las tres primeras líneas corresponden a la interacción con el usuario para dar el nombre de la imagen de entrada, de salida y el mensaje a insertar. El resultado de esta ejecución deberá ser una nueva imagen en disco, con nombre `salida.ppm`, que contendrá una imagen similar a `original.ppm`, ya que visualmente será igual, pero ocultará la cadena “`;Hola mundo!`”.

Observe que la imagen de salida no incluye la extensión, ya que deberá ser `pgm` si la imagen de entrada está en formato `PGM` y `ppm` en caso de que sea `PPM`. Además, el mensaje corresponde a una línea, es decir, deberá leer una cadena de caracteres hasta el final de línea.

Lógicamente, esta ejecución corresponde a un caso con éxito, ya que si ocurre algún tipo de error, deberá acabar con un mensaje adecuado. Por ejemplo, en caso de que la imagen indicada no exista o tenga un formato desconocido.

#### Revelar

El programa para revelar un mensaje oculto realizará la operación inversa al anterior, es decir, deberá obtener el mensaje que previamente se haya ocultado con el programa *ocultar*. Un ejemplo de ejecución podría ser el siguiente:

```

prompt> revelar
Introduzca la imagen de entrada: salida.ppm
Revelando...
El mensaje obtenido es:
¡Hola mundo!
prompt>

```

donde la primera línea corresponde a la interacción con el usuario. Observe que hemos usado la misma imagen que se ha obtenido en la ejecución anterior, y el resultado ha sido exitoso al obtener el mensaje que habíamos ocultado.

De nuevo, tenga en cuenta que si la ejecución encuentra un error, deberá terminar con el mensaje correspondiente.

### Restricciones de diseño

Para resolver estos problemas, será necesario cargar en memoria imágenes —vectores-C de bytes— y cadenas de caracteres. En la solución que proponga no se podrá usar memoria dinámica ni punteros, es decir, será necesario que declare los vectores que necesite como vectores de tamaño fijo, ya sea para almacenar los bytes de la imagen o los caracteres del mensaje.

Como ejemplo, si deseamos cargar una imagen con un tamaño máximo de 1.000.000, podemos incluir el siguiente código:

```

int main()
{
    const int MAXBUFFER= 1000000;
    const int MAXNOMBRE= 100;
    char nombre_imagen[MAXNOMBRE];
    unsigned char buffer[MAXBUFFER];

```

donde puede ver que también hemos añadido una *cadena-C* —para el nombre de la imagen— que contendrá como mucho 100 caracteres. Para esta práctica, podemos limitar el tamaño máximo de la imagen a 1.000.000 bytes, el nombre de una imagen a 100 bytes, y el tamaño máximo de un mensaje a 125.000 bytes.

Lógicamente, cuando indicamos 1.000.000 nos referimos al número total de bytes que ocupa la imagen, ya sea en grises o en color. Por ejemplo, una imagen  $1.000 \times 1.000$  de grises ocuparía el 100% de ese espacio, mientras que una  $1.000 \times 1.000$  en color no cabe, ya que necesitaría el triple de espacio al requerir 3 bytes para cada píxel.

### 3.4.3 Práctica a entregar

El alumno deberá empaquetar todos los archivos relacionados en el proyecto en un archivo con nombre `esteganografia.tgz` y entregarlo en la fecha que se publicará en la página web de la asignatura.

Tenga en cuenta que no se incluirán ficheros objeto ni ejecutables. Es recomendable que haga una “*limpieza*” para eliminar los archivos temporales o que se pueden generar a partir de los fuentes.

Para simplificarlo, el alumno puede ampliar el archivo `Makefile` para que también se incluyan las reglas necesarias que generen los dos ejecutables correspondientes. Tenga en cuenta que los archivos deben estar distribuidos en directorios. Por consiguiente, lo más sencillo es que comience con la estructura de directorios y archivos que ha descargado desde la página y añada lo necesario para completar el proyecto.

Para realizar la entrega, en primer lugar, realice la limpieza de archivos que no se incluirán en ella. Una vez que ha eliminado los archivos generados y sólo tiene los fuentes, sitúese en la carpeta superior —en el mismo nivel de la carpeta `esteganografia`— para ejecutar:

```

prompt> tar zcvf esteganografia.tgz esteganografia

```

tras lo cual, dispondrá de un nuevo archivo `esteganografia.tgz` que contiene la carpeta `esteganografia`, así como todas las carpetas y archivos que cuelgan de ella.

# 4

## Matriz de booleanos

Introducción.....	35
Condiciones de desarrollo	
Problema a resolver.....	35
Ejemplos de ejecución	
Formato de archivos	
Diseño propuesto .....	37
Módulo MatrizBit	
Comprobando la abstracción	
Práctica a entregar .....	42

### 4.1 Introducción

Este proyecto práctico tiene como objetivo principal que el estudiante sea capaz de entender la importancia de ocultar la representación de un nuevo tipo de dato. Se plantea como un problema que intenta mostrar conceptos tan importantes y avanzados como la abstracción y encapsulación mientras plantea el uso de herramientas de programación más básicas, evitando la memoria dinámica y las clases. Los objetivos son múltiples:

1. Mostrar la importancia de la ocultación de información.
2. Practicar con el uso de estructuras, vectores y matrices.
3. Practicar con algunas operaciones básicas de cadenas de caracteres.
4. Practicar con operaciones de E/S básicas y el redireccionamiento de la E/S.
5. Practicar con operaciones a nivel de bit.
6. Ilustrar las ventajas de la programación en base a un diseño que garantiza la independencia de los módulos.

#### 4.1.1 Condiciones de desarrollo

La solución del guión estará basada en el uso de los tipos básicos del lenguaje que ya existían en C: estructuras, *vectores-C*, *cadenas-C* y *matrices-C*. Por tanto, deberá evitar el uso de tipos de la *STL*. Además, se debe abordar el problema evitando la complejidad del uso de memoria dinámica.

El resultado es un programa que ilustra la abstracción sin usar ninguna clase, es decir, un tipo de dato definido con **class** que incluye una parte privada. De esta forma, podrá evaluar la dificultad de garantizar una abstracción sin la ayuda del compilador. Aunque le parezca algo artificioso, podría ser una caso más real si usa un tipo de lenguaje como C.

Una vez terminado el proyecto, debería ser capaz de entender la importancia y comodidad que se deriva de un lenguaje que nos ayuda a diseñar módulos que garantizan la encapsulación.

### 4.2 Problema a resolver

El alumno debe crear un programa **calcular** que permita realizar cálculos simples con matrices booleanas. Una matriz booleana  $M_{F \times C}$  es una estructura bidimensional de  $F$  filas y  $C$  columnas que almacena datos de tipo booleano. Por tanto, un elemento  $m_{ij}$  tendrá dos posibles valores: **true** o **false**.

Las operaciones que se podrán realizar son:

- Operaciones unarias:
  - **NOT**  $M_{F \times C}$ : obtiene una nueva matriz  $R_{F \times C}$  con valores  $r_{ij} = ! m_{ij}$
  - **TRS**  $M_{F \times C}$ : obtiene una nueva matriz  $R_{C \times F}$  con valores  $r_{ij} = m_{ji}$
- Operaciones binarias:
  - $M_{F \times C}$  **AND**  $N_{F \times C}$ : obtiene una nueva matriz  $R_{F \times C}$  con valores  $r_{ij} = m_{ij} \&& n_{ij}$ .
  - $M_{F \times C}$  **OR**  $N_{F \times C}$ : obtiene una nueva matriz  $R_{F \times C}$  con valores  $r_{ij} = m_{ij} || n_{ij}$ .

La lectura de las matrices se podrá realizar tanto desde la entrada estándar como desde un fichero. La salida se realizará en la salida estándar. El formato de la llamada al programa será:

**calcular**    **OPERACIÓN**    [Parámetros]

donde la operación es una de las anteriores (*NOT*, *TRS*, *AND*, *OR*) y los parámetros corresponden a los nombres de los archivos con los que trabajar. En concreto, éstos parámetros podrán ser:

- Si la operación es unaria, cero o un nombre de archivo. Si no hay parámetros, la matriz se leerá desde la entrada estándar. En caso de dar un nombre de archivo, éste contendrá la matriz con la que operar.
- Si la operación es binaria, cero, uno o dos nombres de archivos. Si no hay parámetros, las matrices a operar se leerán desde la entrada estándar. Si hay un único nombre, corresponderá al archivo que contiene el segundo parámetro, mientras que el primer parámetro se obtendrá desde la entrada estándar. Finalmente, si hay dos nombres, éstos corresponderán a los archivos con las matrices que hay que operar.

#### 4.2.1 Ejemplos de ejecución

La mejor forma de mostrar este comportamiento es mediante ejemplos de ejecución. La primera de ellas consiste en una operación **AND** de una matriz consigo misma. Como sabe, el resultado será la misma matriz. El siguiente resultado:

```
prompt> calcular AND ejemplo1.mat ejemplo1.mat
4 6
0 1 0 1 0 1
0 1 0 0 0 0
0 0 1 0 0 0
0 0 0 1 0 0
```

corresponde a la matriz que escribe el programa **calcular** y nos permite conocer lo que tiene el archivo **ejemplo1.mat**. Una ejecución con un operador unario puede ser la siguiente:

```
prompt> calcular NOT exemplo1.mat
4 6
1 0 1 0 1 0
1 0 1 1 1 1
1 1 0 1 1 1
1 1 1 0 1 1
```

donde puede ver que corresponde a la negación de la matriz anterior. Podemos encadenar las dos ejecuciones en una misma línea para realizar una operación **AND** de una matriz con su inversa. El resultado, como sabe, es la matriz con valores cero:

```
prompt> calcular NOT exemplo1.mat | calcular AND exemplo1.mat
4 6
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
```

Observe que el programa se ha llamado dos veces. La primera realiza la negación de la matriz y el resultado, en lugar de presentarlo en consola, se reconduce como entrada estándar a la segunda ejecución. La segunda ejecución realiza una operación **AND** entre lo que entra en la entrada estándar y el parámetro obtenido.

Otra forma de ejecutar el programa que nos muestra el uso de **cat** y otro operador es:

```
prompt> cat exemplo2.mat | calcular OR exemplo2.mat
4 6
1 0 0 0 0 0
0 1 0 0 0 0
0 0 1 0 0 0
0 0 0 1 0 0
```

que también nos revela el contenido del archivo, ya que la operación con la misma matriz da como resultado la identidad.

Una forma distinta de usar **cat** y un operador binario es la siguiente:

```
Consola _ X

prompt> cat ejemplo1.mat ejemplo2.mat | calcular AND
4 6
0 0 0 0 0 0
0 1 0 0 0 0
0 0 1 0 0 0
0 0 0 1 0 0
```

donde puede observar que el operador requiere dos matrices. Al no existir ningún parámetro adicional, ambas se leen desde la entrada estándar.

Un doble encauzamiento que también nos sirve para ilustrar el efecto de la operación TRS es el siguiente:

```
prompt> cat ejemplo1.mat ejemplo2.mat | calcular AND | calcular TRS
6 4
0 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
0 0 0 0
0 0 0 0
```

que corresponde a la matriz traspuesta de la que hemos mostrado en el ejemplo anterior.

## 4.2.2 Formato de archivos

El formato de los archivos es el mismo que el mostrado en los ejemplos de salida anteriores. En concreto, el formato estará compuesto por:

1. Dos valores enteros, filas y columnas, que corresponden a las dimensiones de la matriz. Estarán separados por un “espacio blanco”.
  2. Tantos dígitos binarios como datos tenga la matriz. Estos datos estarán separados de las dimensiones por un “espacio blanco”. Como los dígitos binarios son un único carácter, éstos podrán aparecer sin separadores, incluso en una única línea.

Por otro lado, podemos añadir más versatilidad al formato de estos archivos haciendo que la lectura también sea compatible con un tipo de formato que no requiera indicar las dimensiones. En concreto, podremos especificar una matriz como un número indeterminado de líneas de caracteres de forma que cada una de ellas sea una fila. Usaremos el carácter 'X' para especificar un uno y el carácter '.' para indicar un cero. Un ejemplo de contenido y uso de este archivo es el siguiente:

```
Consola _ X

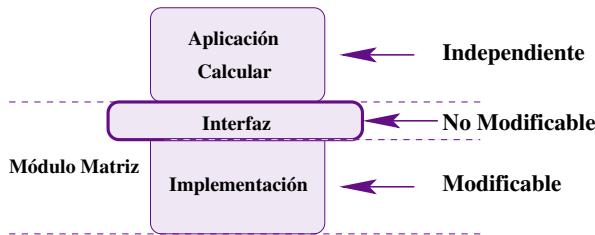
prompt> cat ejemplo3.mat
XXXXXXXXXX
XXXXX.....
.....
prompt> cat ejemplo3.mat | calcular NOT
3 10
0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 1 1
1 1 1 1 1 1 1 1 1
```

Observe que la primera línea muestra en la consola el contenido del archivo compuesto de caracteres que indican unos y ceros. La segunda línea usa este fichero como entrada para calcular una operación lógica. Como puede ver, la salida se realiza siempre en el formato numérico.

Tenga en cuenta que la lectura debe leer líneas independientes y que cada línea debe tener exactamente el mismo tamaño. Si no fuera así, fallaría la lectura.

## 4.3 Diseño propuesto

El programa que vamos a crear está compuesto fundamentalmente de dos módulos, el primero será el que implementa un nuevo tipo de dato *MatrizBit* que contiene una matriz booleana y el segundo contendrá la función **main** que implementa el programa *calcular*. En la figura 4.1 se muestra un esquema en el que se refleja el acceso del módulo de cálculo por medio de una interfaz.



**Figura 4.1**

El diseño que se propone pretende enfatizar la importancia de encapsular los detalles de la representación de forma que el desarrollo de nuevos módulos sea independiente de las estructuras de datos que hemos seleccionado. Aunque la forma más natural de realizar este encapsulamiento es usar una clase con `class`, donde se indica una parte encapsulada con la palabra clave `private`, en este guión proponemos el uso de estructuras para crear el tipo `MatrizBit`.

El obligar un diseño basado en **struct** donde empaquetamos una serie de datos y evitamos añadir ninguna función miembro tiene como objetivos:

- Practicar con el uso de estructuras. Especialmente recomendable si el alumno tiene poca experiencia y necesita afianzar conocimientos básicos antes de avanzar en el diseño de clases.
  - Descubrir que el concepto de encapsulamiento no está asociado únicamente a la programación dirigida a objetos. Es importante darse cuenta de que se puede ocultar la información incluso en lenguajes más simples como C<sup>1</sup>.

El inconveniente de usar una implementación con `struct` será que no tendremos ninguna ayuda del compilador. Si conseguimos aislar los detalles de la representación del resto de módulos será consecuencia de un trabajo disciplinado. Somos nosotros los que nos imponemos la necesidad de ignorar los detalles internos. Tenga en cuenta que si en algún momento dado accedemos a un detalle interno de la estructura, el compilador lo aceptará sin problemas, ya que como sabemos todos los campos son públicos y accesibles sin restricciones.

A pesar de ello, para confirmar que nuestro desarrollo ha sido correcto, vamos a demostrar que hemos conseguido encapsular la representación de una forma muy simple: cambiándola. Efectivamente, si al cambiar la representación un módulo no necesita ninguna modificación, demuestra que el módulo es independiente.

### 4.3.1 Módulo *MatrizBit*

La sintaxis de la interfaz del módulo *Matriz* (véase figura 4.1) que implementa el nuevo tipo es la siguiente:

```

struct MatrizBit {
    // ... Representación de una matriz
};

bool Inicializar(MatrizBit& m, int fils, int cols);
int Filas (const MatrizBit& m);
int Columnas( const MatrizBit& m);
bool Get(const MatrizBit& m, int f, int c);
void Set(MatrizBit& m, int f, int c, bool v);

bool Leer(std::istream& is, MatrizBit& m);
bool Escribir(std::ostream& os, const MatrizBit& m);

bool Leer(const char nombre[], MatrizBit& m);
bool Escribir(const char nombre[], const MatrizBit& m);

void Traspuesta(MatrizBit& res, const MatrizBit& m);
void And(MatrizBit& res, const MatrizBit& m1, const MatrizBit& m2);
void Or(MatrizBit& res, const MatrizBit& m1, const MatrizBit& m2);
void Not(MatrizBit& res, const MatrizBit& m);

```

Como puede notar en este código, no hemos indicado ningún detalle sobre cuál será la representación del tipo *MatrizBit*. Nuestro programa de cálculo se debería poder implementar sin saber qué hemos incluido dentro de la estructura. Sólo es necesario conocer esta sintaxis y el efecto de la llamada a cada operación<sup>2</sup>.

El conjunto de operaciones que hemos listado para el tipo *MatrizBit* se ha presentado en dos bloques diferenciados de 5 y 8 operaciones:

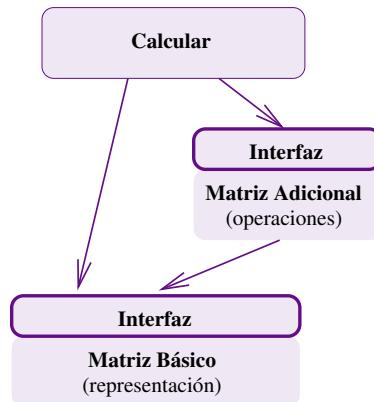
1. El primer grupo lo forman las operaciones fundamentales. Se deben implementar en base a los detalles de la representación. Si cambiamos la representación, deberemos modificarlas para que se adapten a los cambios.
  2. El segundo grupo son operaciones que podría considerarse básicas, pero en la práctica también se podrían implementar en base al primer grupo. Eso significa que podemos escoger entre implementarlas en base a la representación o no.

Para enfatizar los efectos positivos de la encapsulación, vamos a maximizar el nivel de encapsulamiento. La consecuencia de esta decisión es que una modificación en la representación será más simple, pues afectará a menos código. El esquema de módulos que antes presentamos se puede detallar ahora como muestra la figura 4.2.

---

<sup>1</sup>Realmente en lenguajes más simples también podríamos recurrir a otras herramientas que garantizan el encapsulamiento, aunque no es tema de este curso.

<sup>2</sup>En el material adjunto a este guión encontrará la especificación de cada operación.



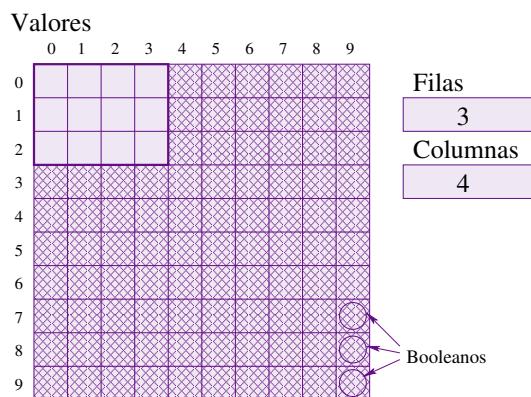
**Figura 4.2**

Si piensa un momento sobre la implementación del segundo grupo de operaciones se dará cuenta de que podría implementarlas ahora mismo sin necesidad de conocer lo que hay dentro de la estructura *MatrizBit*.

## Representaciones del tipo *MatrizBit*

En esta práctica el alumno tendrá que implementar el módulo básico del tipo *MatrizBit* varias veces, cada una de ellas en base a una representación distinta. Las representaciones serán:

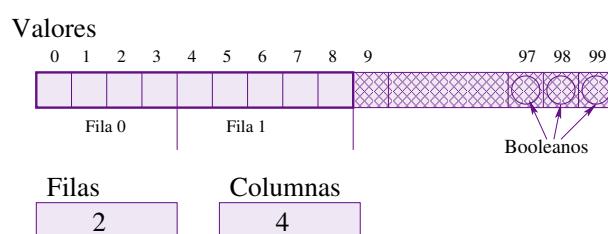
- La primera representación es muy simple, ya que declaramos una *matriz-C* para almacenar los datos booleanos. En la figura 4.3 se presenta la idea gráficamente. En concreto, tendremos que incluir:



**Figura 4.3**

- Una matriz de tamaño  $10 \times 10$  de booleanos. Por lo tanto, no podremos usar una matriz que tenga un número de filas o columnas mayor que 10.
  - Dos enteros que indican el tamaño de la matriz, es decir, las posiciones que realmente se usan en la matriz anterior. En la figura 4.3 usamos la representación para almacenar una matriz de  $3 \times 4$ .

2. La segunda representación intenta mejorar la primera evitando un desperdicio tan grande de memoria. La idea será crear una zona de memoria tan grande como la anterior pero que permita más posibilidades. En la figura 4.4 puede ver la idea gráficamente.



**Figura 4.4**  
Representación basada en un vector.

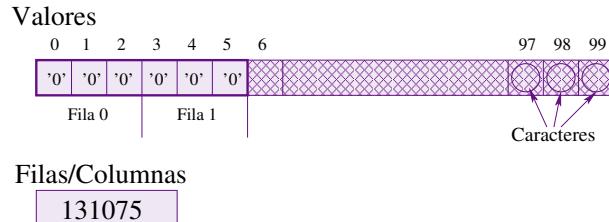
En este caso, hemos cambiado la matriz de tamaño  $10 \times 10$  por un *vector-C* de tamaño 100. Seguramente pensará que desperdiciaremos la misma cantidad de memoria, pero en la práctica esta estructura puede representar muchas más

matrices. Por ejemplo, si queremos una matriz de tamaño  $11 \times 2$  no hay espacio en la primera representación, mientras que en ésta no hay problema.

Por tanto, tendremos que incluir:

- Una vector de tipo **bool** de tamaño 100. Por consiguiente, no podremos usar una matriz que tenga más de 100 posiciones.
- Dos enteros que indican el tamaño, es decir, las posiciones que realmente se usan en la matriz. En la figura 4.4 usamos la representación para almacenar una matriz de  $2 \times 4$ .

3. La tercera representación es muy similar a la anterior, aunque ahora usaremos el tipo **char** en lugar del tipo **bool**. Además, modificaremos la forma en que se almacenan las dimensiones para usar un único entero. Un ejemplo se muestra en la figura 4.5



**Figura 4.5**  
Representación basada en un vector de caracteres.

Observe que hemos añadido un valor de carácter '`'0'`' en cada posición. La figura 4.5 representa el resultado de inicializar una matriz de tamaño  $2 \times 3$  con valores **false**. Como imaginará, si queremos introducir un valor **true** tendremos que insertar el carácter '`'1'`'.

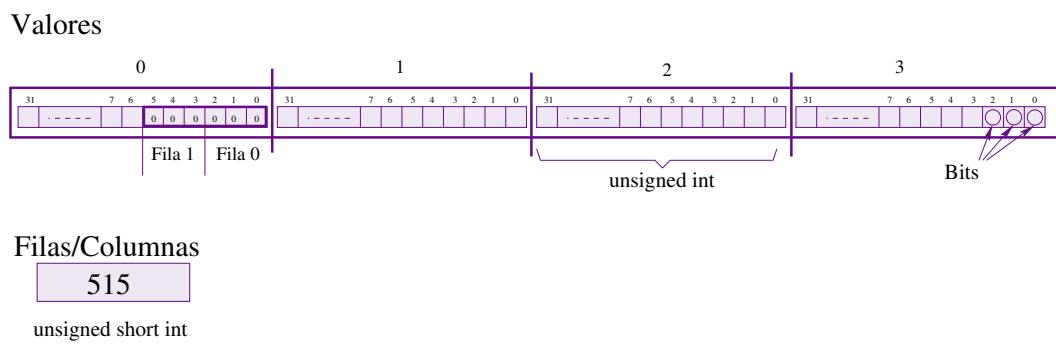
Por otro lado, es probable que le haya sorprendido el valor que se almacena en el entero y que corresponde al tamaño de la matriz. Es el resultado de almacenar en un entero las filas y las columnas. Las filas se almacenarán en los primeros 16 bits, mientras que la columnas se guardarán en los últimos 16 bits.

Por tanto, tendremos que incluir:

- Una vector de tipo **char** de tamaño 100. Al igual que antes, no podremos usar una matriz que tenga más de 100 posiciones.
- Un entero sin signo —**unsigned int**— que indica el tamaño, es decir, las posiciones que realmente se usan como filas consecutivas de la matriz. En la figura 4.5 usamos la representación para almacenar una matriz de tamaño  $2 \times 3$ .

4. La cuarta representación tiene como interés que reduce —del orden de 8 veces— la cantidad de memoria que necesitamos. La idea es que para representar un valor booleano realmente no necesitamos más que un bit. Por tanto, si usamos un **bool** o un **char** realmente podemos estar desperdiciando 7 bits.

Por otro lado, tampoco necesitamos un entero de 32 bits para almacenar las dimensiones. Si nuestras matrices van a tener un tamaño muy pequeño, bastaría con 8 bits para cada valor. En la figura 4.6 se presenta gráficamente cómo se almacenaría una matriz de  $2 \times 3$  elementos, usando los 6 bits menos significativos del primer entero de un vector.



**Figura 4.6**  
Representación basada en bits.

Por tanto, en esta representación tendremos que incluir:

- Una vector de tipo **unsigned int** de tamaño 4. Por consiguiente, no podremos usar una matriz que tenga más de 128 posiciones<sup>3</sup>.
- Un entero sin signo de al menos 16 bits —**unsigned short int**— que indica las dimensiones, es decir, las posiciones que realmente se usan como filas consecutivas.

<sup>3</sup>Suponemos que un entero tiene 32 bits. Se podría calcular como `8*sizeof(unsigned int)`.

## Funciones de E/S y formatos

En la interfaz propuesta se han incluido dos funciones tanto para la lectura como la escritura. Estas funciones deben leer y escribir matrices considerando los formatos que se han especificado en la sección 4.2.2 (página 37). Para implementar la solución, comience implementando las funciones de leer y escribir en un flujo con el formato más simple: valores enteros.

Tenga en cuenta que la función de lectura de matrices con enteros se puede implementar fácilmente usando solamente el operador `>>` de lectura que ya se encarga de eliminar los espacios en blanco. No se pretende dedicar más esfuerzo a la lectura, por ejemplo intentando analizar que los datos se encuentran organizados en filas de tantos elementos como columnas tiene la matriz. Por ejemplo, una matriz como la siguiente sería válida:

```
prompt> cat ejemplo.mat
2 3 1 2 3 4 5 6
```

Por otro lado, no olvide que las funciones de E/S para leer o escribir un fichero pueden implementarse llamando a las funciones que trabajan con flujos —`istream`, `ostream`— por lo que básicamente sólo se tienen que encargarse de abrir los respectivos archivos.

Una vez resuelta la práctica con este formato, puede añadir el nuevo formato de almacenamiento para la lectura, basado en caracteres. Note que es una modificación sobre la anterior, de forma que lo único que tendremos que hacer es modificar la función `Ler` desde un `istream`.

### Sobre nivel de encapsulamiento

La propuesta es realizar la práctica de forma que el *nivel de encapsulamiento* sea muy alto, es decir, haya muy poco código que tenga acceso a los detalles de la representación. Por eso, se propone que el segundo grupo de funciones —las operaciones de más nivel— se implementen en base al primero.

Un mayor nivel de encapsulamiento es positivo porque facilita la modificación. De hecho, este guión está especialmente diseñado para que se dé cuenta programándolo. Sin embargo, también puede descubrir que la implementación y eficiencia de una función puede ser mejor si accede a la representación. Puede intuir una contraposición entre nivel de abstracción y eficiencia<sup>4</sup>.

En la práctica, el diseñador será el encargado de decidir hasta qué punto quiere encapsular una representación y qué módulos quiere incluir como operaciones básicas. Por ejemplo, en esta práctica puede implementar las operaciones `And`, `Or` y `Not` de la cuarta representación directamente accediendo a la representación. Verá que es muy fácil y resulta muy eficiente. La parte negativa de incluir estas funciones en el grupo de las que acceden a la representación es que si cambia algo de la representación, tendrá que revisarlas también.

### 4.3.2 Comprobando la abstracción

El resultado de esta práctica es poco real. En la práctica no se realizan varias implementaciones de un mismo tipo para poder usarlas indistintamente, sino que se selecciona la mejor implementación. Sin embargo, sí es más realista pensar que se implementa una versión con una representación y en el futuro se decide cambiar la representación. Por ejemplo, porque se ha seleccionado una estructura de datos muy simple y rápida para obtener rápidamente una versión operativa del programa, o en el futuro se descubre que los problemas que resuelve el programa son cada vez más complejos y necesitamos una representación altamente optimizada para mejorar la eficiencia.

Para simular esta labor de mantenimiento de un módulo del programa, puede comenzar con implementar la primera representación. Una vez resuelto ese módulo, puede terminar el resto de módulos y completar la aplicación. Debería poder compilar el ejecutable y comprobar que funciona conforme se ha especificado en las secciones anteriores. Una vez que haya comprobado que todo funciona bien, pase a implementar la segunda representación.

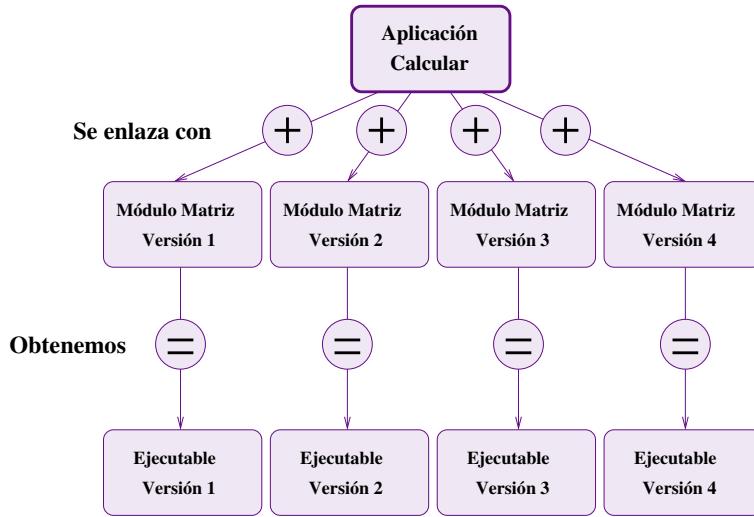
Las modificaciones en la representación le permitirán comprobar si realmente ha respetado la abstracción. Por ejemplo, es posible que cuando cambie la representación no le funcione su código de `main`. Eso indicaría que cuando programó esta función, no lo hizo usando la interfaz del módulo `MatrizBit`, sino accediendo directamente a la representación.

Cuando tenga implementadas las cuatro representaciones, no tendrá cuatro aplicaciones, sino cuatro implementaciones de la misma aplicación. Puede obtener cuatro ejecutables seleccionando la representación correspondiente, pero todos ellos deberían funcionar de la misma forma. En la práctica unos gastan más memoria que otros, unos son más rápidos que otros, pero todos hacen lo mismo. En la figura 4.7 representamos esta idea, donde podemos ver que hay cuatro ejecutables.

### Archivos y selección de la representación

Para facilitar el desarrollo de la práctica y que el estudiante se centre en los conceptos que se quieren mostrar, se ofrece parte de la solución del problema. En concreto, se ofrecen los archivos necesarios para facilitar la compilación y el cambio de representación. Se incluyen los siguientes archivos ya terminados:

<sup>4</sup>De hecho, no es extraño descubrir dos sistemas que realizan la misma función aunque uno es más lento y después de un tiempo sorprendernos porque se ha abandonado precisamente el que parecía más rápido. Si el primero es más fácil de mantener, será más fácil que se adapte a nuevas necesidades.



**Figura 4.7**  
Distintas implementaciones del tipo *Matriz*.

- Archivo **Makefile**. Se ha creado de una forma muy simple, incluyendo una única regla para crear el programa. Cada vez que escriba `make`, se intentará compilar todo el fuente para obtener el ejecutable.  
A pesar de ello, este archivo **Makefile** también nos sirve para compilar un único archivo `cpp`. Cuando quiera generar un archivo objeto, escriba el objetivo a generar y comprobará que se compila con las opciones que hemos incluido. Por ejemplo, si escribe `make matriz_bit.o`, se lanzará la regla implícita correspondiente; este comportamiento es consecuencia de haber usado los identificadores `CXX` y `CXXFLAGS` que la orden `make` conoce.
- Archivos `matriz_bit.h` y `matriz_bit.cpp`. Este es el módulo que implementa las operaciones básicas del tipo *MatrizBit*. Estos archivos están terminados, no hay que tocarlos. Si los revisa, verá que no contiene ninguna de las representaciones, sino que se limita a incluir la representación que selecciona con una macro de precompilación.
- Archivo `matriz_operaciones.h`. Se limita a incluir las cabeceras de las funciones que implementa. Recuerde que estas operaciones no necesitan acceder a la representación, por lo que sólo será necesario implementarlas una vez.

Si desea obtener un ejecutable y probar la aplicación para calcular, debería realizar lo siguiente:

1. Implementar `matriz_bit1.h` y `matriz_bit1.cpp` con la primera representación. Puede compilar y obtener el archivo objeto para resolver los errores de compilación.
2. Implementar `matriz_operaciones.cpp`. Recuerde que ya tenemos resuelto el archivo cabecera. Sólo debe incluir las definiciones de las funciones declaradas. Puede compilar y obtener el archivo objeto para resolver los errores de compilación.
3. Implementar `calcular.cpp`. En este archivo tendrá que incluir los dos archivos cabecera que componen las operaciones con matrices: `matriz_bit.h` y `matriz_operaciones.h`. Con este módulo y lo anterior, ya puede generar el primer ejecutable y depurarlo para que funcione correctamente. También se ofrecen varios archivos de ejemplo de matrices para que le sea más fácil hacer algunas pruebas.

Cuando consiga resolver el problema con esta primera representación, debe realizar la implementación de las otras tres representaciones. Por ejemplo, para implementar y probar la segunda de ellas tendrá que:

1. Implementar `matriz_bit2.h` y `matriz_bit2.cpp` con la primera representación.
2. Cambiar el valor de `CUAL_COMPILO` a 2 en el archivo `matriz_bit.h`.
3. Compilar el proyecto.

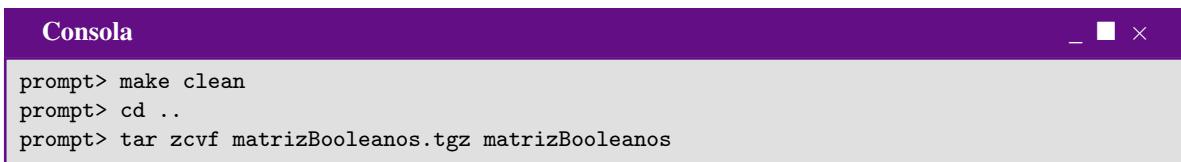
Es posible que obtenga nuevos errores en los otros módulos, es decir, en los que no ha tocado. Estos nuevos errores serán consecuencia de no haber usado correctamente la interfaz de *MatrizBit*. Deberá modificarlos para que no exista este problema cuando vuelva a modificar el valor de `CUAL_COMPILO`.

Una vez terminado todo el proyecto, podrá seleccionar cualquier valor —1, 2, 3 o 4— para indicar la representación deseada. Con un simple `make` obtendrá de nuevo el ejecutable.

## 4.4 Práctica a entregar

Si ha descargado el paquete que contiene los archivos iniciales para realizar la práctica, ya dispone de una carpeta con nombre `matrizBooleanos` donde puede añadir su código. La práctica que debe entregar como resultado del trabajo es el contenido de esta carpeta.

Para poder empaquetar el resultado de este proyecto, es recomendable que realice una “*limpieza*” para eliminar los archivos temporales o que se pueden generar a partir de los fuentes. Una vez eliminados, sítuese en la carpeta superior y use la orden `tar` para obtener el archivo resultado. Más concretamente, ejecute lo siguiente:



The screenshot shows a terminal window titled "Consola". The command history is displayed as follows:

```
prompt> make clean
prompt> cd ..
prompt> tar zcvf matrizBooleanos.tgz matrizBooleanos
```

tras lo cual, dispondrá de un nuevo archivo `matrizBooleanos.tgz` que contiene la carpeta `matrizBooleanos`, así como todos los archivos que cuelgan de ella.



# 5

# Algoritmos con vectores

Introducción.....	45
Objetivos	
Condiciones de desarrollo	
Vectores en memoria dinámica.....	46
Vector dinámico	
Búsqueda	
Otros algoritmos de ordenación	
Rangos de elementos	

## 5.1 Introducción

La práctica con punteros y estructuras en memoria dinámica es fundamental antes de avanzar con nuevos conceptos de C++. Normalmente, los estudiantes con poca práctica suelen tener muchas dificultades en problemas que se resuelven con recursividad y memoria dinámica. Lógicamente, es de esperar que sea así cuando los algoritmos no son triviales y se usan estructuras complejas, sin embargo, la experiencia indica que incluso en problemas que pueden parecer a priori simples —usar aritmética de punteros en una llamada recursiva, por ejemplo— existen dificultades que demuestran que detenerse para practicar con estas herramientas es muy recomendable.

Por otro lado, es importante que los alumnos mejoren la capacidad para crear algoritmos. El problema de diseñar nuevos algoritmos puede parecer complejo para un estudiante novato, siendo más sencillo cuanto mayor es la experiencia. En realidad, muchos de los esquemas que se plantean para resolver un nuevo algoritmo son consecuencia de conocer y practicar con algoritmos clásicos y bien conocidos. Por consiguiente, es importante estudiar algunos problemas, aunque parezcan especialmente difíciles, por la experiencia que permiten adquirir.

El resultado de este guión es un relación de problemas que bien podría ser un tema de teoría de algoritmos en un lenguaje básico como C. A pesar de ello, recuerde que trabajar a este nivel le permitirá adquirir una base de desarrollo fundamental para cuando se enfrente a temas más avanzados de C++.

### 5.1.1 Objetivos

El objetivo principal de este tema es practicar con problemas de programación donde sea necesario trabajar con punteros y vectores reservados en memoria dinámica. Con los problemas propuestos, el alumno debería mejorar su capacidad para manejar:

- Punteros.
- Relación de vectores y punteros, especialmente la aritmética de punteros.
- Reserva y liberación de vectores en memoria dinámica.
- Algoritmos clásicos con vectores.
- Recursividad.

Para ello, se propondrá que el alumno trabaje con problemas conocidos y de especial relevancia de forma que el guón no sólo sirva para conocer aspectos concretos de estos tipos de C++, sino que también tenga un alto contenido en algoritmia.

### 5.1.2 Condiciones de desarrollo

La solución del guón estará basada en el uso de herramientas básicas de C. Cuando se estudie la *STL*, verá que muchas de las dificultades de este tema se resolverán fácilmente mediante tipos de datos de más alto nivel. Sin embargo, la necesidad de practicar con los tipos básicos hace indispensable plantear problemas con la restricción de evitar tipos como `vector<>`, `list<>`, etc.

En este tema usaremos los tipos básicos de C, incluyendo el tipo puntero, la memoria dinámica y las estructuras (`struct`). Recuerde que cuando trabaje con C++ sin ningún tipo de restricción, será más recomendable usar los tipos de más alto nivel que ofrece el lenguaje. En cualquier caso, cuando trabaje a un nivel de abstracción superior, descubrirá que muchos de los conocimientos de este guón le sirven para comprender cómo se podrían haber diseñado e incluso comprender de forma mucho más natural las interfaces que se proponen en la biblioteca estándar.

Es interesante enfatizar que muchos de los problemas y soluciones que mostramos en este tema tienen como objetivo ejercitarse los conocimientos más básicos que necesitaremos en temas posteriores. Un programador de C++ experimentado optaría por diseños más eficaces para resolver la mayoría de los problemas que aquí se presentan.

## 5.2 Vectores en memoria dinámica

La primera parte del guión la vamos a dedicar a la reserva de vectores en memoria dinámica. Recuerde que para reservar un vector, será necesario usar los operadores `new []` y `delete []`, es decir, los operadores con corchete.

Para poder experimentar, vamos a trabajar con datos enteros leídos desde archivo. Como primer ejercicio, el estudiante puede dedicar un momento a estudiar y comprender un programa simple de generación de números. Recuerden que en *GNU/Linux* puede usar la orden `man` con las funciones de la biblioteca estándar de C —por ejemplo, `man atoi`— para obtener una ayuda. El código puede ser el siguiente:

```
#include <iostream>
#include <cstdlib> // rand, atoi
#include <ctime> // time

using namespace std;

int main(int argc, char* argv[])
{
    if (argc!=2) {
        cerr << "Uso: " << argv[0] << " <número de datos>" << endl;
        return 1;
    }

    srand(time(0)); // Inicializamos generador de números

    int n= atoi(argv[1]); // Convertimos cadena-C a int
    for (int i=0; i<n; ++i)
        cout << rand()%n << " "; // %n, por ejemplo
    cout << endl;
}
```

donde puede ver que es un programa que genera una serie de números enteros aleatorios y los envía a la salida estándar. Si lo desea, puede encontrar este programa en el archivo `aleatorios.cpp` con el paquete que habrá descargado con este guión.

Para generar un archivo, no tenemos más que compilar y ejecutar el programa anterior redireccionando la salida hasta un archivo de texto. Por ejemplo, en la siguiente secuencia de órdenes:

```
prompt> aleatorio 15
5 13 3 9 7 14 14 11 13 8 2 8 6 2 14
prompt> aleatorio 15 > datos.txt
prompt> cat datos.txt
3 11 8 0 2 11 8 7 3 7 7 0 10 5 12
```

podemos ver cómo hemos generado 15 números aleatorios para mostrarlos en consola, luego hemos generado otros 15 y guardado en el archivo `datos.txt` y finalmente hemos mostrado su contenido.

### 5.2.1 Vector dinámico

Para esta primera parte, puede trabajar con el archivo `mostrar.cpp` que habrá descargado con este guión. Como primer paso, debería abrir este archivo y estudiar su contenido. Si bien no es un programa terminado, puede observar cómo se ha abstraído el concepto de flujo de entrada para llamar a la función `LeerVecDin` tanto para la entrada estándar como para la entrada desde un archivo.

Un vector dinámico es aquel que está diseñado para poder cambiar de tamaño en tiempo de ejecución. La forma más natural de implementarlo es usando memoria dinámica, es decir, manejando el vector básicamente con un puntero a los datos junto con un entero que indica su tamaño:

```
struct VecDin {
    int* datos;
    int n;
};
```

**Ejercicio 5.1 — Modificando el tamaño del vector.** Implemente una función que reciba un objeto de tipo `VecDin` ya inicializado y modifique el tamaño del vector. Tenga en cuenta que, para ello, probablemente tendrá que reservar un nuevo bloque de memoria. Recuerde que debería mantener los mismos datos (si aumenta el tamaño) o la primera parte de ellos (si disminuye el tamaño). La función tendrá la siguiente cabecera:

```
void ReSize(VecDin& v, int nuevo_tam);
```

*Nota: la tarea aparece en el listado como //FIXME 1.*

### Lectura de un número indeterminado de datos

Una forma de cargar los datos que almacena una vector dinámico es ir añadiendo uno a uno los datos hasta que no haya más. Para eso, podemos crear un vector dinámico que no tenga datos y aumentar su capacidad en uno cada vez que queramos añadir un nuevo dato.

Un ejemplo de esta carga es la lectura de todos los números enteros que hemos almacenado en un fichero de texto como el de la sección anterior. Si leemos un archivo desde la entrada estándar, no sabremos cuántos objetos tenemos que leer hasta que no lleguemos al final. Básicamente, el algoritmo de lectura consiste en “*mientras se lea un dato, añadirlo al final del vector*”.

**Ejercicio 5.2 — Cargando los datos de entrada.** Implemente una función que recibe un flujo de entrada y carga los datos enteros hasta final de entrada. Como resultado, devuelve —con `return`— un nuevo vector dinámico con todos los datos cargados. Use para ello la función que ha implementado en el ejercicio 5.1. La función tendrá la siguiente cabecera:

```
VecDin LeerVecDin(istream& flujo);
```

*Nota: la tarea aparece en el listado como //FIXME 2.*

Estas dos funciones se podrían usar en un programa de prueba que lee un archivo de texto con un número indeterminado de datos y lo muestra en la salida estándar. El código de la función `main` podría ser el siguiente:

```
#include <iostream>
#include <fstream> // ifstream
using namespace std;

// Funciones necesarias para el programa
// ...

int main(int argc, char* argv[])
{
    VecDin v= {0,0};

    if (argc==1)
        v= LeerVecDin(cin);
    else {
        ifstream f(argv[1]);
        if (!f) {
            cerr << "Error: Fichero " << argv[1] << " no válido." << endl;
            return 1;
        }
        v= LeerVecDin(f);
    }

    Mostrar(v,cout);
    Liberar(v); // Libera la memoria dinámica reservada
}
```

**Ejercicio 5.3 — Completar el programa `mostrar.cpp`.** Estudie el código listado en la función `main` anterior para comprender cómo funciona. Añada las funciones `Mostrar` y `Liberar` para probar el programa.

*Nota: la tarea aparece en el listado como //FIXME 3.*

Observe que el resultado del programa anterior debería permitir la siguiente ejecución:

```
Consola
prompt> mostrar datos.txt
3 11 8 0 2 11 8 7 3 7 7 0 10 5 12
prompt> cat datos.txt datos.txt | mostrar
3 11 8 0 2 11 8 7 3 7 7 0 10 5 12
```

### Mejora de la eficiencia

Si revisa el programa de la sección anterior y piensa en las llamadas que se realizan a la función `ReSize`, se dará cuenta de que cada vez que se introduce un dato hay que redimensionar el vector, y cada vez que se redimensiona, hay que copiar todos los datos anteriores. Para hacer más explícito este código, resuelva el siguiente ejercicio.

**Ejercicio 5.4 — Nueva operación: añadir.** Añada una nueva operación `Add` que añade un nuevo dato al final del vector. La cabecera será la siguiente:

```
void Add(VecDin& v, int dato);
```

Para resolverlo, use la función `ReSize` para poder redimensionar el vector y tener un elemento más de capacidad.

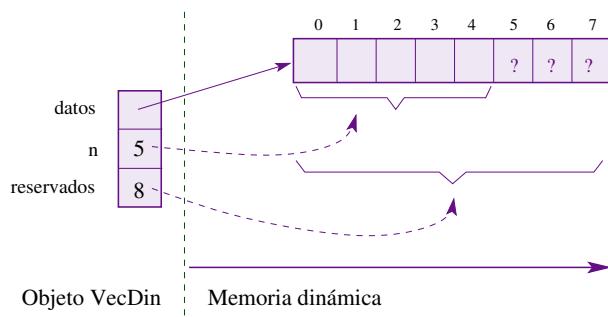
*Nota: la tarea aparece en el listado como //FIXME 4.*

Por ejemplo, si tenemos 1000000 de datos y añadimos uno nuevo, hay que reservar 1000001 posiciones, copiar 1000000 datos anteriores, y añadir el nuevo elemento. Es decir, añadir un elemento al vector es muy ineficiente, por lo que es poco recomendable. Sin embargo, la operación de añadir al final un nuevo elemento es muy útil y probablemente se podría usar en muchas ocasiones. Convendría resolver este problema.

Para mejorar la eficiencia, vamos a cambiar la forma en que se reserva la memoria. Para ello, añadimos un nuevo campo a la estructura `VecDin`:

```
struct VecDin {
    int* datos;
    int n;
    int reservados;
};
```

Los dos primeros objetos miembro son exactamente los mismos, con el mismo significado: el primero es un puntero a la zona de memoria donde están los datos y el segundo nos indica el número de datos que almacenamos. El nuevo campo almacenará los elementos que realmente tenemos reservados en memoria dinámica. La figura 5.1 presenta gráficamente un ejemplo de vector dinámico que tiene cinco datos, aunque el bloque de memoria reservado tiene capacidad para ocho datos.



**Figura 5.1**

Representación de un objeto `VecDin` con cinco datos.

Para resolver el problema, el alumno debe modificar la estructura con el nuevo campo y hacer que la operación `ReSize` tenga en cuenta que:

- Si el nuevo tamaño requerido es menor que el bloque reservado, no tendrá que reservar nueva memoria, sino cambiar el valor de `n`.
- Si el nuevo tamaño requerido es mayor que el bloque reservado, tendrá que volver a reservar memoria.

**Ejercicio 5.5 — Añadir un campo de reservados.** Añada un nuevo campo `reservados` y modifique el código de `ReSize` para que tenga en cuenta que puede haber un número de posiciones reservadas mayor que el de usadas. Tenga en cuenta que pedir un tamaño mayor implica que tendremos que reservar un nuevo bloque de memoria, y pedir un tamaño menor no afecta a la memoria reservada.

Nota: la tarea aparece en el listado como `//FIXME 5`.

A pesar de este cambio, nuestra operación `Add` sigue siendo ineficiente. Realmente hemos añadido la capacidad de disminuir el tamaño y hacer que el número de reservados y usados sea distinto. Sin embargo, las llamadas sucesivas a `Add` no hacen más que aumentar el tamaño, por tanto, siempre tendrá que reservar nueva memoria. Para resolver este problema, vamos a cambiar la implementación de `Add` de la siguiente forma:

- Si el tamaño de usados `n` es menor que el de reservados, se hace un `ReSize` a uno más.
- Si el tamaño de usados `n` es igual al de reservados, se tendrá que reservar el doble del tamaño que se está usando.

Esta nueva estrategia nos resuelve el problema en caso de que las posiciones usadas y reservadas coincidan. Por ejemplo, si tenemos 100 usados de 100 reservados y queremos añadir un elemento, tendrá que reservar un bloque de memoria de 200. Lógicamente, el valor de `n` pasará a 101, dejando 99 huecos para futuras ampliaciones. Si queremos aprovechar la función `ReSize` que tenemos implementada para añadir una posición más al vector, podemos considerar:

- Si el número de usados es menor que el de reservados, basta con hacer `ReSize` a un elemento más. En este caso no hará falta reservar nueva memoria.
- Si tenemos un vector con el número de usados igual al de reservados, una forma muy simple de ampliar el número de reservados al doble es hacer `ReSize` al doble de reservados y luego volver a llamar a `ReSize` a la mitad más uno. La primera llamada amplía la capacidad de reservados y la segunda sitúa el tamaño del vector al valor inicial más uno.

**Ejercicio 5.6 — Mejorar la eficiencia.** Modifique el código de la función `Add` para que duplique el bloque de memoria `reservados` en caso de que no haya más posiciones disponibles para ampliar el número de usados.

Nota: la tarea aparece en el listado como `//FIXME 6`.

Finalmente, vamos a modificar el código para que sea más legible. Para ello, vamos a cambiar el nombre del miembro `n`, que parece poco significativo.

**Ejercicio 5.7 — Renombrar.** Modifique el código del programa anterior para llamar *usados* al miembro que en la versión actual se llama *n*. Si lo desea, puede cambiar el nombre en la estructura y dejar que el compilador genere los errores de compilación que se deriven del cambio.

No piense que este último cambio de nombre ha sido simplemente para hacerle revisar y retocar otra vez todo el código. El hecho de pedir este cambio también tiene como objetivo que se dé cuenta de los problemas que se derivan al modificar los detalles más básicos. Imagine que ha implementado miles de líneas usando la primera versión. Un cambio tan sencillo le obliga a revisar todos los programas que usen esta estructura.

De hecho, incluso es posible que no pudiera cambiarla. Por ejemplo, si ha distribuido su código y lo han usado miles de programadores para crear nuevos programas, todos ellos habrán usado el campo *n*. Si cambia el nombre, ninguno de sus programas volverá a compilar a no ser que se vuelvan a revisar y modificar.

Es interesante que empiece a reflexionar sobre el incremento de dificultad que se deriva del uso de una estructura de datos con la que trabajan múltiples funciones y cuya implementación requiere tener en cuenta muchos detalles interrelacionados. La complicación que se va provocando en este código no es casual: intenta mostrar la forma en que múltiples cambios en código fuertemente acoplado incrementa la dificultad y por tanto disminuye la calidad del software. Imagine esta situación en programas de miles de líneas de código.

### Ejemplo de uso: ordenar por selección

Como ejemplo de uso del tipo *VecDin* que hemos implementado en los ejemplos anteriores, vamos a proponer crear un programa que nos permite mostrar los datos ordenados de un archivo.

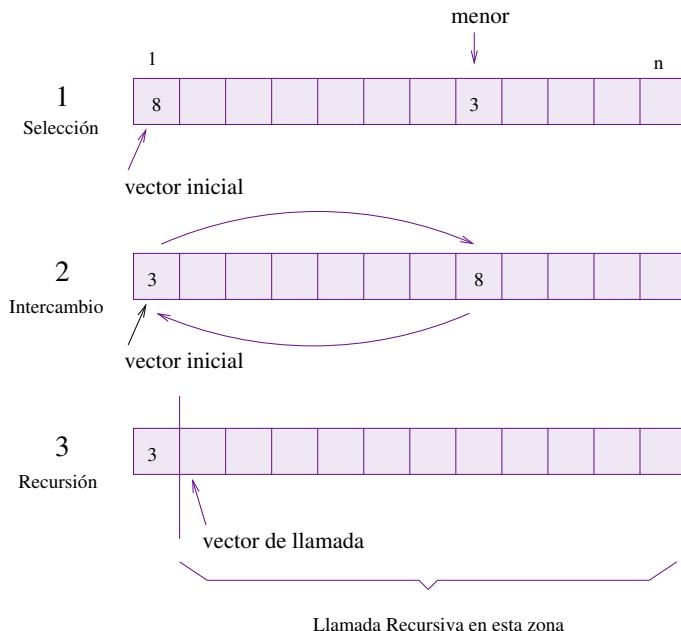
En primer lugar, vamos a resolver el problema de ordenar los elementos de un *VecDin*. Para ello, deberá crear dos funciones:

```
void SeleccionRecursivo(int* v, int n);
void Ordenar(VecDin& v);
```

La segunda tendrá una implementación muy simple, pues se limita a llamar a la primera. La implementación de la primera será un poco más complicada, pues se propone implementar el algoritmo de ordenación por selección. Además, de forma recursiva. Para ello, tenga en cuenta que el algoritmo se puede formular como sigue:

1. Buscar el elemento más pequeño del vector y ponerlo en la posición cero.
2. Ordenar el subvector que empieza en el segundo elemento, el que está en la posición 1.

En la figura 5.2 se muestra la idea gráficamente. El vector de *n* posiciones tiene los elementos desordenados, de forma que el más pequeño es el número 3. El algoritmo busca el menor, lo intercambia con la primera posición y llama a ordenar el subvector descartando este primer elemento.



**Figura 5.2**  
Algoritmo recursivo de ordenación por selección.

Práctica: 5

**Ejercicio 5.8 — Mostrar ordenado.** Modifique el programa anterior que muestra los elementos de un vector incluyendo la posibilidad de ordenarlos. Para ello, el programa tiene en cuenta si se da una opción *-s* que indica que se desea el contenido ordenado. Por ejemplo, algunas llamadas ejemplo son:

```
Consola
prompt> ./mostrar datos.txt
3 11 8 0 2 11 8 7 3 7 7 0 10 5 12
prompt> ./mostrar -s < datos.txt
0 0 2 3 3 5 7 7 8 8 10 11 11 12
prompt> ./mostrar -s datos.txt
0 0 2 3 3 5 7 7 8 8 10 11 11 12
```

Si lo desea, puede usar la función **strcmp** que se encuentra en **cstring**.

*Nota: la tarea aparece en el listado como //FIXME 7.*

Finalmente, es fundamental indicar que en la práctica nadie implementaría el algoritmo de selección recursivamente. Es muy ineficiente, pues requiere un anidamiento de llamadas recursivas del orden del número de elementos a ordenar.

## 5.2.2 Búsqueda

Los algoritmos de búsqueda de un elemento en un vector son un buen problema para practicar con el uso de punteros y aritmética de punteros. Supondremos leído un vector  $v$  de tamaño  $n$  e implementaremos algoritmos de búsqueda sobre él. Para resolver esta parte, puede preparar el archivo **buscar.cpp** que habrá descargado con este guión.

### Generación de un vector aleatorio

Para poder experimentar con un vector de datos, vamos a abordar el problema de generar un vector aleatorio. Para ello, creamos un programa que rellena aleatoriamente los elementos de un vector y los imprime para mostrar el resultado. El programa recibe dos parámetros: el número de elementos a generar y el número máximo que puede contener el vector. Un ejemplo de ejecución del programa puede ser el siguiente:

```
Consola
prompt> ./buscar 15 9
Generados: 2 4 5 6 7 3 6 1 2 3 1 9 4 5 8
```

donde podemos ver que se han generado 15 elementos que tienen como máximo el número 9. Para resolver el problema, el programa debería:

1. Reservar un vector en memoria dinámica con los elementos necesarios.
2. Rellenar el vector con números en el rango solicitado. Por ejemplo, en la posición  $i$  podemos introducir el elemento  $i \% max + 1$ .
3. Barajar aleatoriamente los datos del vector. Podemos generar parejas de índices e intercambiar los elementos que se guardan en las respectivas posiciones.

Parte del programa que resuelve este problema es el siguiente:

```
#include <iostream>
#include <cstdlib> // rand, atoi
#include <ctime> // time

using namespace std;

// Genera un valor del intervalo [minimo,maximo]
int Uniforme(int minimo, int maximo)
{
    double u01= std::rand() / (RAND_MAX+1.0); // Uniforme01
    return minimo + (maximo-minimo+1) * u01;
}

// FIXME: Rellena el vector con n enteros del 1 a max y los mezcla
Generar

int main(int argc, char* argv[])
{
    if (argc!=3) {
        cerr << "Uso: " << argv[0] << " <número de datos> <máximo dato>" << endl;
        return 1;
    }

    srand(time(0)); // Inicializamos generador de números

    int n= atoi(argv[1]);
    if (n<5) {
        cerr << "Debería especificar al menos 5 elementos" << endl;
        return 2;
    }
    else {

        // FIXME
```

```

Generar(v, n, max);
cout << "Generados: ";
for (int i=0; i<n; ++i)
    cout << v[i] << " ";
cout << endl;

// FIXME
}
}

```

Si lo desea, puede encontrar este programa en el archivo `buscar.cpp` con el paquete que habrá descargado con este guión.

**Ejercicio 5.9 — Completar el programa `buscar.cpp`.** Estudie el código que hemos listado. Complete el programa para que funcione conforme hemos indicado.

*Nota: la tarea aparece en el listado como //FIXME 1.*

### Búsqueda secuencial

El algoritmo de búsqueda secuencial recorre los elementos del vector de uno en uno y devuelve la posición donde se encuentra el elemento buscado. Si no se encuentra, devolverá una posición fuera del vector. Una posible implementación es la siguiente:

```

int Buscar(const int* v, int n, int dato)
{
    for (int i=0; i<n; ++i)
        if (v[i] == dato)
            return i;
    return -1;
}

```

Observe que la función devuelve un índice  $-1$  en caso de que el elemento no se encuentre en el vector. Por otro lado, observe que hemos indicado con `const` que el vector no se puede modificar.

**Ejercicio 5.10 — Localizar la primera ocurrencia.** Modifique el programa del ejercicio 5.9 añadiendo un trozo de código después de imprimir los elementos generados. Este trozo de código preguntará por el valor de un `dato`, lo buscará usando la función anterior, e imprimirá la posición donde se encuentra. Un ejemplo de ejecución es:

```

prompt> ./buscar 15 9
Generados: 8 7 3 2 5 9 3 6 5 2 4 6 1 4 1
Introduzca un dato a buscar: 7
Encontrado en: 1

```

*Nota: la tarea aparece en el listado como //FIXME 2.*

Esta función es válida para localizar la primera ocurrencia de un elemento del vector `v` que tiene `n` elementos. Suponga que deseamos obtener todas las ocurrencias. Por ejemplo, con el siguiente efecto:

```

prompt> ./buscar 15 9
Generados: 5 4 3 1 7 5 8 2 1 6 3 4 2 9 6
Introduzca un dato a buscar: 5
Encontrado en: 0
Encontrado en: 5

```

Podemos proponer una modificación del programa para que pudiéramos pasar como parámetro el subvector que se sitúa después de la posición localizada. El código propuesto es el siguiente:

```

int pos=Buscar(v, n, dato);
while (pos!=-1) {
    cout << " Encontrado en: " << pos << endl;
    pos= Buscar(v+pos+1, n-pos-1, dato);
}

```

**Ejercicio 5.11 — Fallo en la búsqueda de todas las ocurrencias.** Pruebe el trozo de código anterior para localizar todas las ocurrencias de un entero `dato`. ¿Por qué falla? Una vez descubierto el porqué del fallo anterior, modifique el código haciendo que la función reciba siempre el vector `v`, pero añadiendo otro parámetro que indica el índice desde el que buscar. Complete el código para que funcione.

*Nota: la tarea aparece en el listado como //FIXME 3.*

Finalmente, es interesante destacar que podríamos implementar la función de búsqueda con otra interfaz. Por ejemplo, podemos devolver el tamaño  $n$  en caso de que el elemento no se encuentre en el vector. Note que esta posición es la que está después de la última, y por tanto fuera del vector.

**Ejercicio 5.12 — Posición después de la última.** Modifique el programa anterior para que la función de búsqueda devuelva el tamaño del vector en caso de no encontrar el elemento.

Nota: la tarea aparece en el listado como `//FIXME 4.`

### Búsqueda secuencial garantizada

Un algoritmo de búsqueda secuencial ligeramente distinto es aquél que se aplica sobre un vector en el que está garantizado el elemento. Es decir, sabemos que el resultado de la búsqueda va a ser una posición del vector. En este caso, no es necesario iterar sobre el vector con una condición que incluye el número de elementos total, es decir, podemos simplificar la condición de iteración.

**Ejercicio 5.13 — Búsqueda garantizada.** Modifique el programa anterior creando una nueva función de búsqueda con la siguiente cabecera:

```
int BuscarGarantizada(const int* v, int dato);
```

Esta función tiene como precondición que el dato está en el vector. Por tanto, la primera búsqueda seguro que tendrá éxito. En este algoritmo, tendremos en cuenta que iteramos sólo mientras que no localicemos el elemento, ya que sabemos que el éxito de la búsqueda está garantizado.

Nota: la tarea aparece en el listado como `//FIXME 5.`

El problema de esta última solución es que no podemos garantizar que el usuario pregunte por un dato del vector. Imagine que se equivoca e introduce un dato incorrecto que la función no localiza. El programa falla con un error de ejecución. Tal vez piense que la culpa es del usuario, sin embargo, no es así. Realmente, el programa tiene un error, pues el programador de la función `main` no ha respectado la especificación. Si la función tiene una precondición, el que usa la función debe garantizar que se cumple.

Recordemos que una función que tiene una precondición sólo terminará correctamente si se cumple dicha condición. Es decir, si no se cumple, la función podría dar lugar a cualquier comportamiento, incluso una terminación anormal del programa.

**Ejercicio 5.14 — Cumpliendo las precondiciones.** Modifique el programa anterior para que se cumpla la precondición. Para ello, el programa debe reservar  $n+1$  elementos, aunque el usuario seguirá trabajando con los  $n$  primeros. Antes de la llamada a la búsqueda, se introduce el elemento a buscar una posición detrás de la última, para garantizar que la función se encontrará con el elemento en una posición reservada.

Nota: la tarea aparece en el listado como `//FIXME 6.`

Finalmente, vamos a proponer una nueva interfaz para la función de búsqueda. En esta nueva versión de la función de búsqueda, vamos a aprovechar la aritmética de punteros como una forma cómoda y eficiente de acceder secuencialmente a las posiciones de un vector. En concreto, se propone una función como la siguiente:

```
int* BuscarGarantizada(int* inicial, int dato);
```

En esta función, pasamos la posición donde debe comenzar la búsqueda y se devuelve un puntero a la posición donde está el elemento. Observe que es una búsqueda garantizada, por lo que no tenemos que indicar nada sobre el tamaño del vector.

**Ejercicio 5.15 — Búsqueda con punteros.** Modifique el programa anterior —en el que añadimos el elemento después de la última posición— para buscar todas las ocurrencias del dato buscado. Para ello, debe implementar la función anterior sin usar el operador corchetes `[ ]` de acceso a las posiciones del vector. Recuerde que la búsqueda terminará por devolver un puntero a la posición donde hemos añadido el elemento extra.

Nota: la tarea aparece en el listado como `//FIXME 7.`

### Búsqueda binaria o dicotómica

La búsqueda binaria es mucho más rápida que la secuencial. Sin embargo, tiene una importante limitación: necesita que el vector esté ordenado. En la práctica, cuando necesitemos buscar un elemento, deberíamos optar por este algoritmo.

El problema es que si tenemos una serie de datos sin ordenar, sería necesario aplicar previamente un algoritmo de ordenación, que es mucho más lento que una simple búsqueda. A pesar de ello, tenga en cuenta que si tenemos que realizar múltiples búsquedas en un mismo conjunto de datos, puede resultar una mejor opción dedicar un tiempo a un paso previo de ordenación.

Para poder trabajar con datos ordenados, vamos a añadir un algoritmo de ordenación a nuestro programa. Inicialmente, optaremos por uno bastante simple, aunque no especialmente eficiente: el *algoritmo de inserción*. Recuerde que este algoritmo consiste en mantener una primera parte del vector ordenada e ir insertando nuevos elementos hasta que el vector queda totalmente ordenado. Por ejemplo, si tenemos  $p$  elementos ordenados, del 0 al  $p-1$ , podemos insertar el elemento de índice  $p$

en la posición que le corresponde de forma que ya tenemos  $p+1$  elementos ordenados. Si aplicamos esta idea repetidamente, al final tendremos todos ordenados.

**Ejercicio 5.16 — Ordenación por inserción.** Añada al programa anterior una función *OrdenarInsercion* que recibe un vector de enteros y ordena los elementos usando el algoritmo de ordenación por inserción. Para probarlo, imprima los datos generados y ordenados antes de llamar a la búsqueda. Un ejemplo de ejecución del programa es:

```

prompt> ./buscar 15 9
Generados: 5 4 1 2 1 8 3 6 7 4 9 2 6 5 3
Ordenados: 1 1 2 2 3 3 4 4 5 5 6 6 7 8 9
Introduzca un dato a buscar: 5
    Encontrado en: 8
    Encontrado en: 9
  
```

Probablemente disponga de algún listado con este algoritmo implementado. Intente volver a pensarlo e implementarlo. No consulte la solución hasta que lo haya resuelto o si considera que está tardando demasiado tiempo.

*Nota: la tarea aparece en el listado como //FIXME 8.*

El algoritmo de búsqueda binaria o dicotómica es muy intuitivo: para buscar en una secuencia ordenada, consultamos el elemento central, si el buscado es más pequeño seguimos buscando en la primera mitad, si no, seguimos buscando en la segunda mitad. La implementación más habitual es mediante un bucle que itera delimitando la zona donde se encuentra el elemento:

```

int BusquedaBinaria (const int* vec, int n, int dato)
{
    int izq= 0, der= n-1;

    while (izq<=der) {
        int centro= (izq+der) / 2;
        if (vec[centro] > dato)
            der= centro-1;
        else if (vec[centro] < dato)
            izq= centro+1;
        else return centro;
    }
    return -1;
}
  
```

Observe que la función termina cuando los índices *izq* y *der* se cruzan —no está el elemento— pero también cuando al consultar el centro encuentra directamente que es el buscado, es decir, devuelve la llamada en medio del bucle.

Por otro lado, podemos implementar la función de búsqueda binaria como un algoritmo recursivo. Tenga en cuenta que la búsqueda binaria consiste en consultar el elemento central y si no corresponde al elemento buscado, aplicar la misma idea en un vector de menor tamaño.

**Ejercicio 5.17 — Búsqueda binaria recursiva.** Modifique la implementación de la función anterior de búsqueda binaria pero usando un algoritmo recursivo. Mantenga la misma cabecera pero con nombre *BusquedaBinariaRec*. Cuando la haya terminado, añada al programa anterior un trozo de código que pregunta por un elemento y devuelve si lo ha localizado y la posición donde está. Un ejemplo de ejecución es:

```

prompt> ./buscar 15 9
Generados: 5 2 3 4 8 9 4 7 6 2 3 1 5 6 1
Ordenados: 1 1 2 2 3 3 4 4 5 5 6 6 7 8 9
Introduzca un dato a buscar: 1
    Encontrado en: 0
    Encontrado en: 1
Introduzca un dato a buscar binario: 9
    Encontrado en: 14
  
```

Observe que en primer lugar hemos preguntado por el primer elemento y después por el último. Realice esa prueba cuando lo ejecute, así como la de preguntar por un elemento que no se encuentra en el vector.

*Nota: la tarea aparece en el listado como //FIXME 9.*

## Búsqueda binaria con interpolación

La *búsqueda por interpolación* es una forma de búsqueda binaria que pretende bajar el número de consultas o “cortes” que aplicamos al vector. En la práctica no será generalmente necesario, pues la búsqueda binaria ya es muy eficiente. Sin

embargo, en casos donde haya muchos datos y además sea costoso realizar la consulta de uno de ellos (imagine la búsqueda en un archivo) podemos plantear la interpolación.

La idea es muy intuitiva y seguro que ya lo ha aplicado en su vida real. Por ejemplo, imagine que tiene que buscar en un diccionario —en papel— una palabra. Es posible que esté pensando en la palabra “*aburrido*”. Seguro que si selecciona una página para determinar si la palabra está en la primera parte o en la segunda, intentará que sea una página más bien del principio del diccionario. Obviamente, una palabra que empieza por “*ab*” debería estar más bien en esa zona.

La diferencia con la búsqueda binaria es que la interpolación nos permite hacer una estimación de dónde puede estar el elemento que buscamos. En lugar de consultar el elemento central, calculamos la posición aproximada de dónde está y cortamos el vector en dos trozos que podrían incluso ser de tamaños muy dispares.

La forma de cortar el vector es suponiendo que la distribución de elementos es más o menos uniforme y por tanto una interpolación lineal debería de caer muy cerca del elemento buscado. Como estamos trabajando con enteros, la posición que buscamos se puede calcular como:

$$pos = izq + \frac{dato - vec[izq]}{vec[der] - vec[izq]} * (der - izq) \quad (5.1)$$

**Ejercicio 5.18 — Búsqueda binaria con interpolación.** Implemente una función *BusquedaBinariaInterp* que implemente la búsqueda binaria con interpolación. Para ello, use la función iterativa de la sección anterior y modifíquela para calcular la posición “central” con la ecuación 5.1. Cuando lo haga —especialmente si no le funciona— recuerde que los elementos son enteros y que además puede haber repetidos.

Nota: la tarea aparece en el listado como //*FIXME 10*.

Por último, es interesante indicar que la búsqueda binaria se puede implementar de forma recursiva o iterativa. Sin embargo, desde el punto de vista de la eficiencia es mejor la iterativa, ya que no es necesario anidar llamadas en la pila y una simple vuelta al principio —iterando con el bucle— permite volver a aplicar un nuevo paso con poco esfuerzo. Si piensa en la relación entre los dos algoritmos, se dará cuenta de que el recursivo se puede clasificar como un caso de *recursividad de cola* que es fácil de transformar en iterativo.

### 5.2.3 Otros algoritmos de ordenación

En las secciones anteriores se han incluido algunos ejercicios relacionados con los algoritmos de ordenación por selección e inserción. Un tercer algoritmo básico muy conocido es el *algoritmo de la burbuja*. Como sabe, este algoritmo consiste en hacer una comparación entre cada dos elementos consecutivos desde un extremo del vector hasta el otro, de forma que el elemento de ese extremo queda situado en su posición final. Una posible implementación es la siguiente:

```
void OrdenarBurbuja (int vec[], int n)
{
    for (int i=n-1; i>0; --i)
        for (int j=0; j<i; ++j)
            if (vec[j] > vec[j+1]) {
                int aux = vec[j];
                vec[j] = vec[j+1];
                vec[j+1]= aux;
            }
}
```

Para esta parte del guión, puede trabajar con el archivo *ordenar.cpp* que habrá descargado con este guión. La versión que puede descargar ya es un programa que puede lanzar. Un ejemplo de ejecución es el siguiente:

Consola

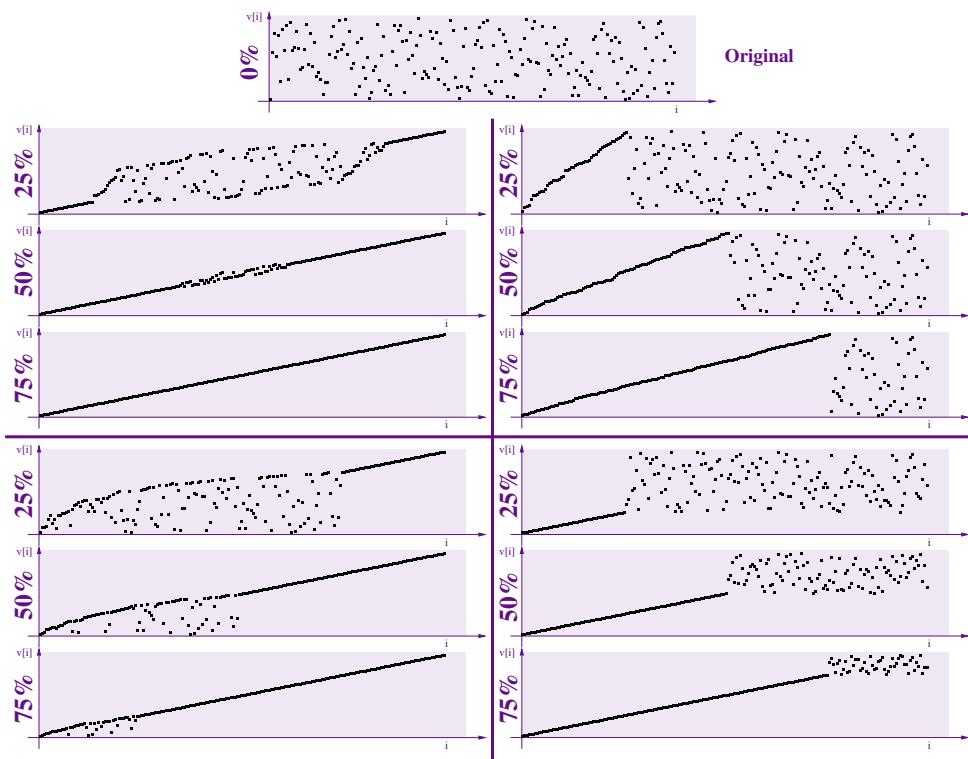
```
prompt> ./ordenar 15 9
Generados: 1 5 2 1 3 8 7 6 4 2 6 5 9 4 3
Ordenados: 1 1 2 2 3 3 4 4 5 5 6 6 7 8 9
```

Si estudia el código, comprobará que básicamente se limita a generar valores aleatorios, ordenarlos con el algoritmo de la burbuja y presentarlos en la salida estándar.

**Ejercicio 5.19 — Burbuja mejorada.** El algoritmo de la burbuja es muy ineficiente. De hecho, se considera el más ineficiente de los básicos. Una posible mejora puede ser controlar si se ha hecho alguna modificación. Por ejemplo, si intentamos ordenar un vector ya ordenado, sería más eficiente controlar que en la primera iteración no se ha hecho ningún cambio, por lo que el algoritmo podría terminar. Modifique el algoritmo implementado en el programa *ordenar.cpp* para que incluya esta comprobación.

Nota: la tarea aparece en el listado como //*FIXME 1*.

En la figura 5.3 se han presentado gráficamente 4 algoritmos de ordenación clásicos. Entre ellos, están el algoritmo de inserción, burbuja y selección. Observe que en la parte superior hemos dibujado el vector original; en el eje horizontal se representa la posición del vector y en el vertical el contenido. La altura del punto indica si el valor es mayor o menor.



**Figura 5.3**  
Ordenación al 25 %, 50 %, 75 % de algoritmos clásicos.

**Ejercicio 5.20 — Distinguir algoritmos.** Piense detenidamente en los 4 modelos presentados en la figura 5.3. Intente distinguir los 3 que corresponden a los algoritmos de inserción, burbuja y selección.

El cuarto modelo presentado en la figura 5.3 corresponde al algoritmo de burbuja bidireccional. Lo puede encontrar como *shaker sort* o *cocktail sort* en inglés. Consiste en una modificación del algoritmo clásico de ordenación de la burbuja. Si en éste se “desliza” un elemento desde un extremo del vector hacia su posición final, en el bidireccional se aplica este traslado hacia la derecha y hacia la izquierda de forma alternativa. Es decir, si el primer paso desliza el mayor elemento hacia la parte derecha, el segundo desplaza el menor hacia la izquierda. Si vuelve a revisar la figura anterior, descubrirá fácilmente el efecto que tiene esta estrategia.

**Ejercicio 5.21 — Burbuja bidireccional.** El programa `ordenar.cpp` implementa el algoritmo clásico de la burbuja. Modifique el programa para realizar una ordenación bidireccional, incluyendo la comprobación de estado implementada en el ejercicio 5.19.

*Nota: la tarea aparece en el listado como //FIXME 2.*

No es objetivo de este curso analizar y comparar los distintos algoritmos. Si está interesado, puede consultar la bibliografía —por ejemplo Knuth [7] o Sedgewick [8]— para un estudio más formal. En principio, nos limitaremos a decir que son algoritmos simples e inefficientes.

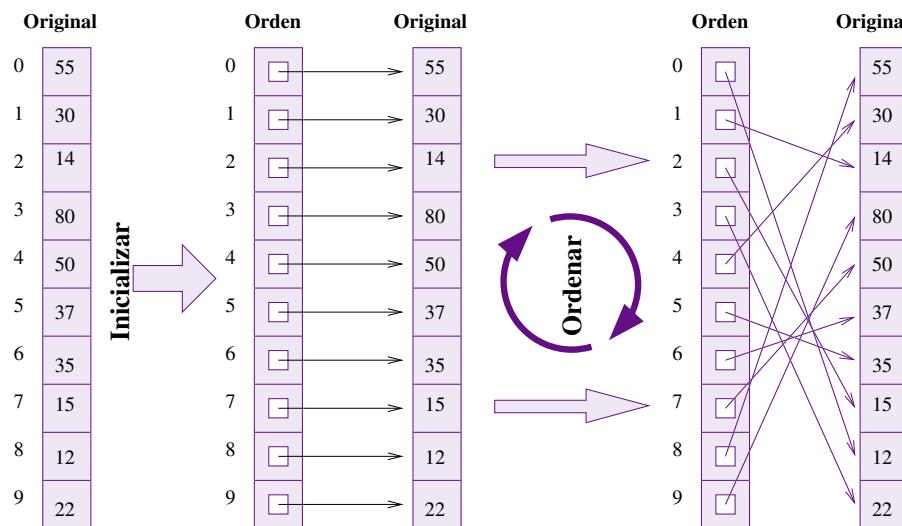
### Ordenar apuntadores

En los algoritmos de ordenación que se han presentado en las secciones anteriores, se ha modificado el vector moviendo los datos de unas posiciones a otras. Es posible ordenar los elementos evitando copiar o mover los elementos. En el caso en que no podamos modificar el vector o cuando mover los datos sea una operación costosa, podemos usar apuntadores para organizarlos.

Una primera opción simple y eficaz es crear un vector de índices de localización de los elementos y ordenar los índices en lugar de los datos. En esta sección, vamos a plantear una solución equivalente, pero ordenando punteros en lugar de índices.

En la figura 5.4 puede ver un ejemplo de ordenación sin modificar los datos originales. Observe que los datos siguen en la misma posición después de ordenar. Lo que hemos modificado ha sido un vector de punteros —representados con un cuadro blanco— que se sitúan en un vector que controla el orden. Observe que el primer puntero apunta al número 12 —el más pequeño— mientras que el último puntero apunta al número 80 —el más grande—.

**Ejercicio 5.22 — Ordenar apuntadores.** En este ejercicio deberá realizar dos tareas:



**Figura 5.4**  
Ordenación indirecta con punteros.

- Modifique el programa `ordenar.cpp` del ejercicio 5.21 para incluir una nueva función de ordenación que reciba un vector de punteros, un vector de datos —que no se pueda modificar— y un número de elementos. El efecto de la función es que el vector de punteros se ordenará conforme al orden del vector de datos. Puede usar cualquier algoritmo de ordenación.

*Nota: la tarea aparece en el listado como //FIXME 3.*

- Modifique la función `main` añadiendo un trozo de código para reservar un vector de punteros adaptado al tamaño, ordenar los punteros —use la función anterior— y listar los elementos ordenados. Después de esto, terminará con el listado del vector original y el resultado de ordenar el vector original con el algoritmo que teníamos implementado. Finalmente, debe listar cuántas posiciones se ha desplazado cada dato de la posición original. Un ejemplo de ejecución es el siguiente:

**Consola**

```
prompt> ./ordenar 15 9
Ordenados:  1   1   2   2   3   3   4   4   5   5   6   6   7   8   9
Generados:  7   5   4   9   1   1   8   4   2   6   5   2   6   3   3
Ordenados:  1   1   2   2   3   3   4   4   5   5   6   6   7   8   9
Salto   :  -4  -4  -6  -8  -9  -9  -1   5  -2   8  -2   2  12   7  11
```

Observe que el número 8 se generó en la posición de índice 6 y ha terminado en la posición 13. Por tanto, se ha desplazado 7 posiciones. Los saltos negativos corresponden a desplazamientos hacia la izquierda.

*Nota: la tarea aparece en el listado como //FIXME 4.*

Finalmente, suponga que dos elementos son idénticos y consecutivos en el vector de generados. ¿Tendrán el mismo valor de salto? Razone la respuesta.

### Parametrizando el orden

En todos los ejemplos mostrados hasta ahora hemos supuesto que el vector está ordenado de menor a mayor. Además, no hay duda de cuál es la forma de ordenar dos enteros. Sin embargo, en la práctica podemos estar interesados en otro orden. Por ejemplo, de mayor a menor.

Para crear un algoritmo genérico podemos introducir un nuevo parámetro<sup>1</sup> que indique el orden de los elementos. En nuestro caso, podemos pasarle la función que indica el orden que hay entre dos enteros, es decir, un parámetro adicional con un puntero a función que calcula el orden. Por ejemplo, podemos crear la siguiente función:

```
int OrdenHabitual(int l, int r)
{
    return l-r;
}
```

La función se ha diseñado para devolver un número negativo si el primero es menor que el segundo, un positivo si es al contrario, o un cero en caso de que sean iguales. Con esta función, podemos indicar al algoritmo de ordenación `OrdenarBurbuja` que queremos un orden de menor a mayor. Si queremos cambiarlo, podemos usar otra función:

```
int OrdenHabitualInversa(int l, int r)
```

<sup>1</sup>En este guión nos limitaremos a un algoritmo de bajo nivel basado en punteros a función. Este método es válido también en C. Más adelante, aprenderá más y mejores métodos para parametrizar la función en C++.

```
{
    return r-1;
}
```

para obtener el orden inverso. Note que a pesar de cambiar el orden, la función *OrdenarBurbuja* es exactamente la misma. Sólo cambiamos la llamada pasándole la nueva función.

**Ejercicio 5.23 — Parametrizar el orden.** Modifique el programa que ha obtenido en el ejercicio 5.22 para parametrizar el algoritmo de ordenación por burbuja. Para ello, introduzca un nuevo parámetro puntero a función que permita recibir cualquiera de las dos funciones anteriores. Ejecute el programa con ambas funciones para comprobar que funciona correctamente.

Una vez comprobado el correcto funcionamiento del programa, analice la siguiente función y deduzca el orden que obtendrá en la salida.

```
int Orden(int l, int r)
{
    return (l&1 && r&1) || ((l&1)==0 && (r&1)==0) ? r-1 : (l&1)-(r&1);
```

Si no consigue deducirlo, pruébela en el programa y vuelva a analizarla considerando lo que ha obtenido.

Nota: la tarea aparece en el listado como //FIXME 5.

## Ordenando cadenas

Un ejercicio especialmente interesante es el de ordenar *cadenas-C*. En este caso proponemos crear una estructura bidimensional que corresponda a una serie de cadenas a ordenar. Esta estructura no es rectangular, ya que las cadenas tienen longitudes independientes, determinadas por el carácter final de cadena. Crearemos un programa que usa el algoritmo de ordenación *Shell* para leer una serie de cadenas y mostrarlas ordenadas.

No es objetivo de este documento estudiar el algoritmo *ShellSort*, aunque el lector ya tiene algunas ideas útiles para entenderlo, puesto que no es más que la aplicación repetida del algoritmo de inserción, aunque en cierto subconjunto de elementos del vector. Nos limitaremos a reescribir un algoritmo resuelto para el tipo entero y adaptarlo al tipo cadena de caracteres.

Para resolver esta sección, dispone de los siguientes archivos:

- Un archivo *cadenas.txt*. Almacena varias líneas para probar los algoritmos.
- Un archivo *shellsort.cpp*. Contiene un programa que genera una serie de números enteros y los presenta ordenados. Para ello, llama al algoritmo *ShellSort*. Es un programa terminado que el alumno usará como solución de referencia para el programa a resolver.
- Un archivo *ordenar\_cadenas.cpp*. Es el archivo que tendrá que resolver.

El programa final que deseamos obtener corresponde al programa *ordenar\_cadenas*, que lee una serie de cadenas desde la entrada estándar o desde un archivo para presentarlas ordenadas. Un ejemplo de ejecución es el siguiente:

```
Consola
prompt> ./ordenar_cadenas cadenas.txt
Original:
Hola
Adiós
Prueba
prueba, aunque con minúscula
Mayúscula
minúscula

Resultado:
Adiós
Hola
Mayúscula
Prueba
minúscula
prueba, aunque con minúscula
```

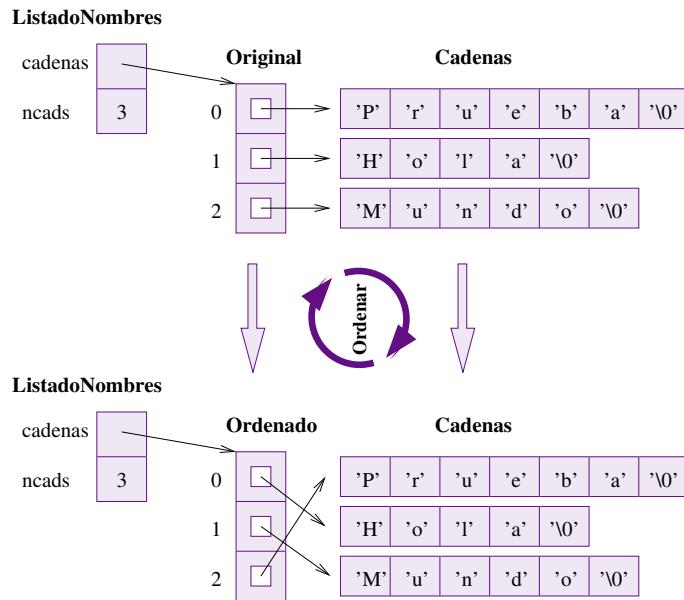
Práctica: 5

Para ello, tendrá que resolver varias funciones propuestas en el archivo *ordenar\_cadenas.cpp* que puede descargar junto con este guión. Tenga en cuenta que el programa ya está medio resuelto, ya que tiene la función *main* terminada, y tiene la referencia del archivo *shellsort.cpp* que contiene el código para ordenar enteros. El programa, básicamente, consiste en:

1. Cargar las líneas de un archivo.
2. Mostrar las líneas leídas.
3. Ordenar las líneas.

4. Mostrar el resultado.
5. Liberar la estructura de memoria dinámica.

La carga de las líneas del archivo consiste en crear una estructura como la mostrada en la figura 5.5. Esta estructura contiene un puntero y un entero. El puntero apunta a una zona de memoria dinámica desde la que cuelgan las cadenas reservadas. El algoritmo de ordenación consiste en modificar los punteros del vector que contiene todas las cadenas. En la parte inferior de la figura, puede observar el resultado de ordenar el vector usando la función `strcmp`. Note que sólo hemos tenido que modificar el valor de los punteros, es decir, ordenar ese vector para obtener un listado de nombres ordenado.



**Figura 5.5**  
Ordenación de cadenas-C

**Ejercicio 5.24 — Ordenar cadenas.** Complete el archivo `ordenar_cadenas.cpp` para que realice la tarea de crear una estructura dinámica con las cadenas correspondientes a las líneas de un flujo, ordenarlas, mostrarlas y liberarlas. Para ello, tendrá que seguir las instrucciones que aparecen en el archivo.

Nota: las tareas aparecen en el listado como `//FIXME 1 a //FIXME 5`.

## 5.2.4 Rangos de elementos

En las secciones anteriores hemos presentado múltiples ejemplos en los que manejamos los elementos de un vector mediante un puntero a la primera posición, devolvemos la posición de un elemento mediante el puntero que lo apunta o incluso la inexistencia de un elemento con un puntero que apunta a una posición no válida (después de la última).

La aritmética de punteros y la relación tan estrecha entre éstos y los *vectores-C* permite desarrollar una amplia gama de algoritmos de una forma muy simple y eficiente. En esta sección, vamos a presentar de una forma un poco más formalizada el concepto de *rango de elementos*. En principio, lo detallaremos en el contexto de los punteros y los vectores, aunque más adelante podrá estudiar la generalización del concepto en C++ descubriendo una herramienta potente y eficiente para resolver problemas tanto a bajo nivel como con los contenedores y tipos más avanzados de la *STL*.

**Definición 5.1 — Rango.** Un rango es una secuencia de elementos de un contenedor que está delimitada por dos posiciones que indican, respectivamente, el primer elemento y el elemento siguiente al último.

Dado un vector `vec` que contiene  $n$  elementos, hablamos de *rango de elementos* a cualquier subsecuencia de elementos que contiene el vector. La forma de especificar cualquiera de ellas podría basarse en indicar los índices de los elementos incluidos. Sin embargo, en C++, el rango se especificará mediante dos punteros. Esta forma de delimitar una secuencia de elementos nos permite manejarlos sin ninguna referencia al contenedor inicial —el vector `vec` o el tamaño `n`— al que corresponden<sup>2</sup>.

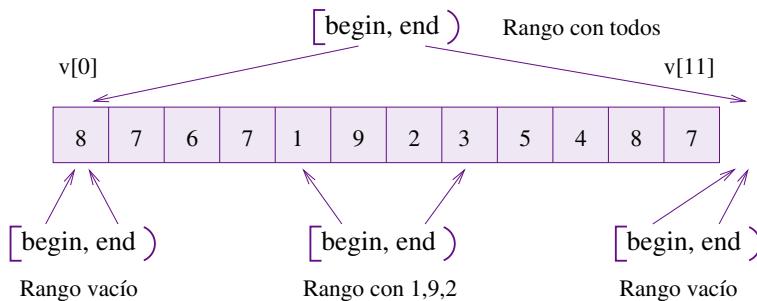
En el caso de un vector `vec` de  $n$  elementos podemos declarar dos punteros que determinan un rango que abarca todos ellos:

```
int* begin= vec;
int* end= vec+n;
```

<sup>2</sup>Más adelante estudiará la generalización de estos rangos a otros contenedores y comprenderá cómo podemos usar distintas formas de recorrerlos simplemente con dos “apuntadores”.

Observe que todos los elementos que hay en el vector están en el rango `[begin, end)`. Note que especificamos el rango con los símbolos `[]` para enfatizar que el elemento apuntado por `begin` es parte del rango, mientras que el apuntado por `end` no lo es. En el ejemplo anterior, el valor `vec+n` apunta después del último elemento.

En la figura 5.6 presentamos varios ejemplos, incluyendo el vector completo, un subvector y el rango vacío. Note que todos los valores de `begin` o `end` son punteros que indican un elemento del vector o el elemento después del último. Un ejemplo de listado de elementos en un rango `[begin, end)` es el siguiente:



**Figura 5.6**  
Ejemplos de rangos en un vector.

```
for (int* p= begin; p!=end; ++p)
    cout << *p << endl;
```

Observe que no hemos tenido que hacer ninguna referencia al vector de elementos que estamos recorriendo. Es decir, podemos controlar cualquier subsecuencia de un vector con una pareja de punteros que definen un rango. Incluso una *secuencia vacía* en caso de que los dos punteros sean idénticos.

En las siguientes secciones proponemos una serie de ejercicios para trabajar con rangos y aritmética de punteros. Para ello, use el archivo `rangos.cpp` que habrá descargado con este guión. Como primer paso, debería abrir este archivo y estudiar su contenido. Es muy parecido a los programas anteriores, aunque incluimos un nuevo algoritmo simple de ordenación que comentamos más adelante.

### Búsqueda secuencial en un rango

La búsqueda secuencial de un elemento en un vector se puede realizar con una interfaz basada en un rango. La idea consiste en pasar un par de punteros que indican el comienzo y fin del rango. Lógicamente, si queremos hacer una búsqueda en todo el vector podemos pasarlo como comienzo el vector y como fin la posición después del último elemento del vector. El resultado será un puntero al elemento que buscamos o el mismo punto fin del rango para indicar que no lo hemos localizado.

Además, podemos usar la misma interfaz para otras ocurrencias del elemento buscado. Lo único que tenemos que hacer es indicar el rango que comienza justo después de la posición que acabamos de encontrar.

**Ejercicio 5.25 — Búsqueda secuencial en un rango.** Modifique el programa `rangos.cpp` para incluir un trozo de código con el que buscar todas las ocurrencias de un dato. Para ello, por un lado tendrá que completar una función de búsqueda basada en una interfaz adecuada y, por otro, escribir el código que incluye un bucle que itera para localizar todas las posiciones. Un ejemplo de ejecución podría ser el siguiente:

```
Consola
prompt> ./rangos 15 9
Generados: 8 1 5 4 7 2 1 2 9 3 4 6 5 3 6
Introduzca un dato a buscar: 1
    Encontrado en: 1
    Encontrado en: 6
Ordenados: 1 1 2 2 3 3 4 4 5 5 6 6 7 8 9
```

Observe que el resultado de la búsqueda corresponde a las posiciones en el vector, aunque la función de búsqueda nos devolverá un puntero al elemento encontrado.

Nota: la tarea aparece en el listado como `//FIXME 1.`

**Práctica: 5**

### Ordenación de los elementos de un rango

La ordenación de los elementos de un vector también puede implementarse con una interfaz basada en rangos. Para poder mostrar algún ejemplo, vamos a trabajar con un nuevo algoritmo de ordenación: el *algoritmo del gnomo* (*gnome sort* en inglés). El nombre surge porque la idea está basada en la forma en que un gnomo ordena las macetas por tamaño.

La idea consiste en que el gnomo recorre las macetas de izquierda a derecha mientras que estén bien ordenadas, es decir, mientras que la siguiente sea mayor o igual que la actual. En cuanto encuentra una más pequeña, la intercambiamos con la

anterior, lo que hace que volvamos hacia atrás para seguir intercambiándola hasta encontrar su posición correcta. Una vez situada en su sitio, volvemos a avanzar hacia delante y repetir los mismos pasos. El algoritmo acabará cuando al avanzar hacia delante comprobando que están bien ordenadas, llegamos al final de la fila de macetas.

En la práctica, es un algoritmo que tiene mucho en común con el algoritmo de inserción, pues mantiene un subvector inicial ordenado en el que va insertando el siguiente elemento en la secuencia. Lo que lo hace distinto es el método para insertar el siguiente elemento, basado en intercambios, que recuerda en gran medida a la forma en que el algoritmo de la burbuja desplaza un elemento.

Como resultado, es un algoritmo inefficiente que se podría clasificar junto con los algoritmos básicos que hemos presentado. Lo que lo hace especialmente interesante es su simplicidad. El hecho de que podamos movernos en una posición hacia delante y hacia detrás permite crear un código que con un sencillo bucle resuelve el problema. En el programa `rangos.cpp` puede revisar un ejemplo de implementación.

**Ejercicio 5.26 — Ordenar un rango.** Modifique el programa `rangos.cpp` incluyendo una nueva función `OrdenarGnom` —quedará sobrecargada— que recibe como parámetros un rango a ordenar. Deberá añadir un trozo de código en la función `main` que ordene `v2` con esta función y presente el resultado. Un ejemplo de ejecución podría ser el siguiente:

```
Consola
prompt> ./rangos 15 9
Generados: 3 5 4 8 7 1 9 6 6 2 1 2 5 3 4
Introduzca un dato a buscar: 6
    Encontrado en: 7
    Encontrado en: 8
Ordenados: 1 1 2 2 3 3 4 4 5 5 6 6 7 8 9
Segundo v: 1 2 6 6 7 8 5 2 9 3 5 4 3 4 1
Ordenados: 1 1 2 2 3 3 4 4 5 5 6 6 7 8 9
```

Observe que las dos secuencias de ordenados son idénticas. Esto es de esperar, pues hemos generado los mismos números, aunque los hemos barajado de distinta forma.

*Nota: la tarea aparece en el listado como //FIXME 2.*

## Transformación de rangos

Podemos usar una pareja de punteros que especifican un rango para implementar funciones que no sólo usan los elementos, sino que operan con ellos. Por ejemplo, podemos crear una función genérica para realizar una transformación de cada uno de los elementos de un rango. Esta función podría tener tres parámetros: los dos primeros indican el rango y el tercero es un puntero a función que recibe un dato y lo transforma.

**Ejercicio 5.27 — Transformar un rango.** Modifique el programa `rangos.cpp` incluyendo una función `Transformar` que recibe un rango un puntero a función para recorrer todos los elementos y aplicar dicha función a cada uno de ellos. Para probarla, añada un trozo de código a `main` para transformar la segunda secuencia de datos ordenados. Un ejemplo de ejecución es el siguiente:

```
Consola
prompt> ./rangos 15 9
Generados: 1 5 7 6 3 2 2 6 5 9 1 3 4 8 4
Introduzca un dato a buscar: 2
    Encontrado en: 5
    Encontrado en: 6
Ordenados: 1 1 2 2 3 3 4 4 5 5 6 6 7 8 9
Segundo v: 4 2 6 4 1 5 7 8 3 3 6 5 1 2 9
Ordenados: 1 1 2 2 3 3 4 4 5 5 6 6 7 8 9
El doble: 2 2 4 4 6 6 8 8 10 10 12 12 14 16 18
```

Observe que los elementos de la última línea son exactamente el doble de la línea anterior. Es decir, hemos modificado todos y cada uno de los elementos aplicando la función `Doble`.

*Nota: la tarea aparece en el listado como //FIXME 3.*

Note que la implementación que se ha obtenido no es especialmente eficiente. Debemos tener en cuenta que la transformación de cada uno de los elementos se tiene que realizar a partir de un puntero a función, es decir, es necesario *desreferenciar* y gestionar la llamada. Es claro que una implementación específica en la que se incluya la operación dentro del bucle, prescindiendo del puntero a función, es mucho más eficiente.

A pesar de ello, el resultado que hemos obtenido es una generalización que puede usarse para cualquier operación. Más adelante, podrá estudiar mecanismos de generalización y abstracción mucho mejores: más simples de usar y prácticamente sin perder eficiencia<sup>3</sup>.

Por otro lado, podemos crear otros algoritmos que operen sobre varios rangos. Como ejemplo, proponemos un algoritmo de mezcla. Suponiendo que tenemos dos secuencias de elementos ordenados, se pueden unir en una nueva secuencia ordenada con un algoritmo muy eficiente. La idea consiste en usar dos punteros que apuntan a los primeros elementos de las secuencias de entrada, comparar los elementos para volcar el más pequeño al resultado, y avanzar el correspondiente puntero. Si aplicamos esta idea repetidamente, obtendremos la nueva secuencia ordenada.

**Ejercicio 5.28 — Mezcla de rangos.** Modifique el programa `rangos.cpp` completando la función `Mezclar` que se ha incluido en el listado. Esta función asume que el puntero `begin` apunta a una zona con suficiente capacidad como para incluir todos los elementos de las dos secuencias de entrada. Para probarla, incluya un trozo de código en `main` para crear una zona de memoria suficiente, mezclar las dos secuencias ordenadas que tenemos, y listar el resultado. Un ejemplo de la ejecución es el siguiente:

```
prompt> ./rangos 15 9
Generados: 1 1 6 5 4 2 7 3 8 9 5 4 6 2 3
Introduzca un dato a buscar: 1
    Encontrado en: 0
    Encontrado en: 1
Ordenados: 1 1 2 2 3 3 4 4 5 5 6 6 7 8 9
Segundo v: 1 3 5 8 4 3 9 2 4 2 7 6 1 6 5
Ordenados: 1 1 2 2 3 3 4 4 5 5 6 6 7 8 9
El doble: 2 2 4 4 6 6 8 8 10 10 12 12 14 16 18
Mezclados: 1 1 2 2 2 2 3 3 4 4 4 4 5 5 6 6 6 6 7 8 8 8 9 10 10 12 12 14 16 18
```

Observe que en la mezcla se han incluido los elementos de la primera secuencia con el resultado de transformar al doble la segunda.

Nota: la tarea aparece en el listado como `//FIXME 4.`

<sup>3</sup>Serán de especial interés las nuevas posibilidades de C++11 que incluye en *lambda-cálculo* como una forma muy simple de indicar una operación.



# 6

# Celdas enlazadas

Introducción.....	63
Objetivos	
Condiciones de desarrollo	
Listas de celdas enlazadas .....	64
Búsqueda, inserción y borrado	
Celda controlada desde la anterior	
Ordenación	
Rangos de elementos	

## 6.1 Introducción

Conocer y gestionar correctamente estructuras dinámicas de celdas enlazadas es importante para un curso básico de programación donde se pretenden cubrir conocimientos básicos sobre memoria dinámica. Además, es fundamental si queremos abordar temas más complejos y especializados como el estudio y diseño de estructuras de datos avanzadas.

En este guión presentaremos algunos algoritmos básicos sobre celdas enlazadas, creando situaciones que necesitarán que el estudiante entienda perfectamente cómo se manejan punteros, en general, y que permitan practicar y afianzar los conocimientos sobre punteros a celdas enlazadas, en particular. Todos estos conocimientos serán la base para estudiar y entender temas centrados en las estructuras de datos, donde los detalles de implementación —especialmente el manejo de celdas enlazadas— no deberían ser un obstáculo para el estudiante.

El contenido que se presenta aquí tratará únicamente con celdas simplemente enlazadas. Lógicamente, la posibilidad de crear estructuras con varios campos de tipo puntero permite crear infinidad de situaciones. El objetivo de este documento se limita a que el estudiante entienda y maneje correctamente la reserva de objetos simples enlazados. Diseños más complicados corresponderían, más bien, a un curso de estructuras de datos por lo que se dejarán para cursos posteriores.

Este guión bien podría ser una segunda parte del guión anterior, que recordemos está dedicado a la reserva y manejo de vectores en memoria dinámica. Aunque el tamaño y variedad de los problemas sugiere la división en dos partes diferenciadas.

Podríamos proponer un tercer bloque donde permitiéramos mezclar vectores y celdas enlazadas en complejas estructuras de datos. Sin embargo, por un lado los contenidos de estos dos bloques son suficientes para entender y practicar con las distintas alternativas del curso y, por otro lado, otros problemas deberían abordarse en proyectos de desarrollo concretos o incluso cursos centrados en el estudio de estructuras de datos.

### 6.1.1 Objetivos

El objetivo principal de este tema es practicar con problemas de programación donde sea necesario manejar memoria dinámica. En concreto, entender y superar los problemas propuestos implica que el alumno estaría capacitado para trabajar con:

- Punteros.
- Celdas enlazadas.
- Recursividad.

Para ello, se propondrá que el alumno trabaje con problemas simples y de especial relevancia de forma que el guión no sólo sirva para conocer aspectos concretos de manejo de celdas enlazadas, sino que también tenga cierto contenido en algoritmia.

### 6.1.2 Condiciones de desarrollo

La solución del guión estará basada en el uso de herramientas básicas similares a las de C. Cuando se estudie la *STL*, verá que muchas de las dificultades de este tema se resolverán fácilmente mediante tipos de datos de más alto nivel. Sin embargo, la necesidad de practicar con los tipos básicos hace indispensable plantear problemas con la restricción de evitar tipos como `vector<>`, `list<>`, etc.

En este tema usaremos los tipos básicos de C++, incluyendo el tipo puntero, la memoria dinámica y las estructuras (`struct`). Recuerde que cuando trabaje con C++ sin ningún tipo de restricción, será más recomendable usar los tipos avanzados que ofrece el lenguaje en la *STL*.

Es de especial interés el tipo de dato `list<>` que ofrece el lenguaje para manejar listas en C++98, o incluso más interesante el tipo `forward_list<>` que se añade en C++11. Cuando desarrolle un programa donde necesite una lista de elementos, podrá usar este contenedor que resuelve la mayoría de los problemas de bajo nivel, pudiendo crear una solución simple y eficiente sin complicar sus algoritmos con el manejo de memoria dinámica. Como ocurría en el guión anterior, descubrirá que muchos de los conocimientos de este guión le sirven para comprender cómo se podrían haber diseñado e incluso comprender de forma mucho más natural las interfaces que se proponen en el lenguaje para los tipos `list<>` y `forward_list<>`.

En este guión nos limitaremos a las celdas enlazadas simples, es decir, una celda estará enlazada con la siguiente mediante un puntero, lo que las relaciona más directamente con el tipo `forward_list<>`. En cursos posteriores podrá estudiar estructuras como las celdas enlazadas dobles, que permiten enlazar una celda con la anterior y la siguiente, más directamente relaciona con el tipo `list<>`.

## 6.2 Lista de celdas enlazadas

Las posibilidades de crear estructuras de datos basadas en celdas enlazadas son ilimitadas. El hecho de poder añadir y enlazar distintos objetos nos permitiría proponer infinidad de estructuras. Como hemos indicado, dado que estamos en un curso introductorio, nos limitaremos a trabajar con celdas simplemente enlazadas.

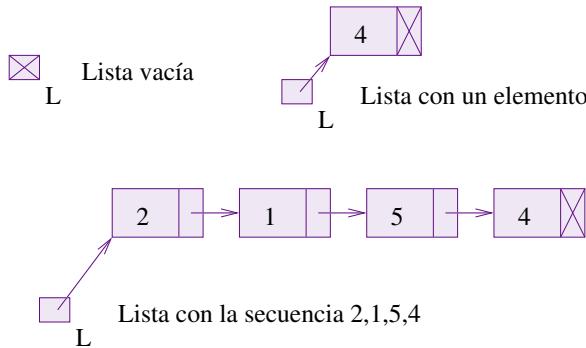
En la figura 6.1 puede ver distintos ejemplos de secuencias de elementos. Para crear estas listas, hemos supuesto un objeto `L` y una estructura de dos campos similar a la siguiente:

```
struct Celda {
    int dato;
    Celda* sig;
};

Celda* L;
```

donde el puntero nulo se ha representado con un aspa. Es importante enfatizar que en una lista de celdas enlazadas:

- Una lista se podrá manejar con un único puntero —en el dibujo con nombre `L`— del que colgarán todos los datos desde el primero al último.
- Una lista vacía se representa con el puntero nulo.
- El último elemento de la lista no vacía viene determinado por una celda que contiene un puntero siguiente nulo.



**Figura 6.1**  
Secuencias de datos con celdas enlazadas.

Para resolver esta parte del guión, puede usar el archivo `celdas.cpp` que habrá descargado con él. Observe que en el archivo ya aparece una lista que está inicializada con el puntero nulo, es decir, está vacía (véase figura 6.1). Este archivo no se puede compilar, pues no incluye la función `Add` que se llama en `main`.

**Ejercicio 6.1 — Compilar el archivo de trabajo.** Complete el archivo `celdas.cpp` para que pueda compilarse. Para ello, debe incluir la función `Add` e implementarla de forma que al llamarla la lista incluya una nueva celda —reservada en memoria dinámica— que contenga el elemento añadido.

*Nota: la tarea aparece en el listado como //FIXME 1.*

Si estudia el código que contiene el archivo, verá que aunque es compilable, no es un programa correcto, ya que reserva recursos de memoria y no los libera.

Realmente, el sistema recuperará la memoria reservada cuando el sistema operativo desaloje el ejecutable. A pesar de ello, un programa debería liberar los recursos que reserva sin delegar la tarea en la finalización del programa. Imagine que quiere reutilizar ese trozo de código en otro programa. Será más fácil reutilizarlo si ya contiene todas las líneas necesarias para que no queden bloques de memoria reservados innecesariamente.

**Ejercicio 6.2 — Liberar la memoria reservada.** Añada una función `Liberar` que recibe una lista con un número indeterminado de celdas y las libera aplicando el operador `delete` a cada una de ellas. La lista deberá quedar vacía,

es decir, quedará con el valor puntero nulo. En la función `main` deberá incluir la llamada para liberar la lista que se ha declarado.

*Nota: la tarea aparece en el listado como //FIXME 2.*

Por otro lado, el programa ejecutable resultado de incluir la función que falta —`Add`— y el código para liberar la lista no tiene ningún efecto visible. Nos hemos limitado a reservar una serie de celdas y liberarlas.

**Ejercicio 6.3 — Listar la lista generada.** Añada una función `Listar` que recibe una lista con un número indeterminado de celdas y muestra los elementos delimitados entre llaves y separados por comas. Un ejemplo de ejecución es el siguiente:

```
prompt> ./celdas 15 9
Lista: {3,2,4,1,7,4,8,8,7,6,7,9,5,3,4}
```

Note que una lista vacía se listará como dos caracteres —las llaves— sin elementos en medio.

*Nota: la tarea aparece en el listado como //FIXME 3.*

Finalmente, vamos a añadir alguna función adicional que nos permitan consultar el estado de la lista. Concretamente, vamos a crear una función para calcular el tamaño de las lista.

**Ejercicio 6.4 — Cálculo del tamaño.** Añada una función `Size` para calcular el tamaño de una lista. La función recibe una lista y devuelve un entero que indica el número de celdas de la lista.

Para probar la función, añada el código necesario en `main` para calcular el tamaño de la lista. Un ejemplo de ejecución es el siguiente:

```
prompt> ./celdas 15 9
Lista: {4,1,7,3,2,8,9,1,2,1,8,5,9,7,1}
La lista contiene 15 elementos.
```

*Nota: la tarea aparece en el listado como //FIXME 4.*

## 6.2.1 Búsqueda, inserción y borrado

La búsqueda en una lista de celdas enlazadas es simple, no porque el código sea más sencillo que en el caso de un vector, sino porque la única posibilidad razonable es un algoritmo secuencial que recorra las celdas desde la primera a la última. Aunque tengamos una lista ordenada, no nos sirve para implementar la búsqueda binaria porque el acceso al elemento central sería ineficiente. El resultado del algoritmo sería tan bueno como hacer la búsqueda secuencial, por tanto, no tiene sentido implementarlo.

En principio, podemos plantear la búsqueda como un algoritmo que localiza la celda donde se encuentra un elemento. Para eso, el algoritmo debe posicionarse en la primera celda y preguntarse por el contenido, si es el elemento buscado, se devuelve el puntero a la celda, si no, se avanza a la siguiente celda. Lógicamente, podemos llegar al final sin encontrarlo; en ese caso podemos devolver el puntero nulo.

**Ejercicio 6.5 — Búsqueda de un elemento.** Escriba una función `Buscar` que localiza la celda de un elemento en una lista enlazada. Para ello, recibe la lista a la primera celda y el elemento a buscar. El resultado es un puntero a la celda localizada o un cero para indicar que no está. Para probarlo, añada al programa `celdas.cpp` con el que está trabajando un trozo de código que pregunta por un dato y responde con un mensaje que indica si el dato está en la lista enlazada. Un ejemplo de ejecución es el siguiente:

```
prompt> ./celdas 15 9
Lista: {2,9,6,1,2,7,6,8,6,5,4,1,5,1,6}
La lista contiene 15 elementos.
Introduzca un dato a buscar: 50
Buscar celda: El dato no está en la lista.
```

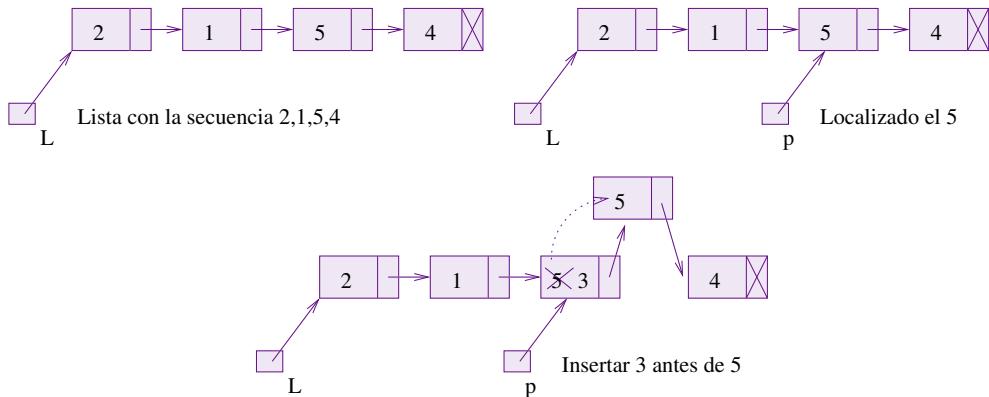
Note que la parte añadida al programa corresponde a las dos últimas líneas. En este caso, hemos preguntado por el número 50, que lógicamente no está en la lista.

*Nota: la tarea aparece en el listado como //FIXME 5.*

Considere el diseño de la función de búsqueda del ejercicio anterior. El resultado de la función es un puntero a la celda que contiene el elemento, aunque podríamos haber diseñado una función que devuelve un booleano. Pensemos un momento en cómo podríamos usar el puntero  $p$  que nos devuelve. ¿Qué operaciones nos permitiría hacer ese puntero?

- Nos permite repetir la búsqueda para buscar la siguiente aparición del mismo elemento. Es decir, si queremos saber si el elemento está más de una vez, podemos buscar la celda para devolver un puntero  $p$  que la localiza. En caso de que no sea cero, podemos volver a buscar el elemento en la lista  $p \rightarrow sig$ .
- Podríamos desenganchar cualquiera de las celdas que hay a continuación. Por ejemplo, es muy simple enganchar una nueva celda a continuación de la encontrada.

Sin embargo, localizar una celda con un puntero  $p$  no nos permite desengancharla de la lista ni insertar una nueva celda en la posición donde está  $p$ . Por ejemplo, imagine que queremos poner un valor entero *nuevo* antes del elemento que hemos encontrado. La única forma de aprovechar el puntero devuelto es crear una celda a continuación, mover el dato encontrado a esa nueva celda y sobreescribir la anterior con el valor *nuevo*.



**Figura 6.2**  
Localizar un elemento e insertar el 3 antes. Versión no recomendada.

En la figura 6.2 puede ver gráficamente la idea de esta inserción. Observe que desde la posición  $p$  podemos acceder al campo  $p \rightarrow sig$ , e insertar una nueva celda. En esta celda movemos el 5 que teníamos y nos queda un nuevo elemento 3 insertado antes. El trozo de código podría ser como sigue:

```
Celda* L;
// ... se rellena L
Celda* p= Buscar(L,5); // Suponemos que está el 5

Celda* aux= new Celda;
aux->dato= 5; // Creamos la nueva con el 5
aux->sig= p->sig; // Hacemos que apunta al 4
p->dato= 3; // Sobreescrivimos el dato con el insertado
p->sig= aux; // Enganchamos la nueva celda
```

Piense un momento en otra operación: el borrado. Imagine que queremos borrar el elemento encontrado. Si piensa en la solución que podemos encontrar, el puntero que nos devuelve la función de búsqueda no nos permite destruir la celda donde está el elemento. Para destruirla, tendremos que desengancharla de la lista. Para desengancharla, tenemos que acceder a la celda anterior. Podríamos pensar en forzar la solución, por ejemplo, copiando el elemento que hay a continuación en la celda localizada y borrando la celda siguiente. Aunque esto no nos serviría para la última celda.

En la figura 6.3 puede ver gráficamente la idea de este borrado. Observe que desde la posición  $p$  podemos acceder al campo  $p \rightarrow sig$  desde el que eliminar la celda siguiente. El trozo de código podría ser como sigue:

```
Celda* L;
// ... se rellena L
Celda* p= Buscar(L,5); // Suponemos que está el 5

Celda* aux= p->sig; // Usamos aux para descolgar la siguiente
p->dato= p->sig->dato; // Movemos el de la derecha (aux->dato)
p->sig= p->sig->sig; // Desenganchamos la celda aux (apuntamos a aux->sig)
delete aux; // Liberamos la celda que sobra
```

Sin embargo, si estamos en la última celda ésta sería una operación incorrecta, pues el puntero  $p \rightarrow sig$  vale cero, no hay ningún elemento que mover.

Por otro lado, las operaciones conllevan mover un objeto de una celda a otra. Normalmente, las operaciones con celdas —ya sea la inserción o el borrado— normalmente corresponden a crear nuevas celdas o borrar las existentes, evitando la

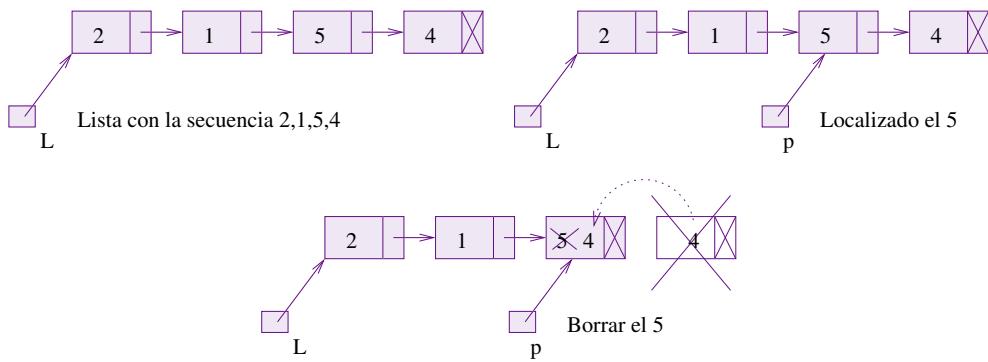


Figura 6.3

Localizar un elemento y borrarlo. Versión no recomendada.

asignación entre celdas<sup>1</sup>. Precisamente es la razón para seleccionar una estructura de este tipo: no tener que mover o desplazar elementos cuando queremos insertar o borrar.

### 6.2.2 Celda controlada desde la anterior

Para resolver las situaciones expuestas en la sección anterior, resulta recomendable controlar la posición de un elemento teniendo el control del puntero de la celda anterior. Por ejemplo, si localizamos una celda mediante un puntero a la celda anterior, podríamos realizar un borrado fácilmente. Note que, para poder desenganchar una celda, tenemos que modificar el puntero que la apunta.

Existen distintas soluciones para poder manejar celdas enlazadas de forma que se pueda modificar la celda anterior de una forma simple y eficiente. En esta sección vamos a proponer un ejercicio que permite trabajar con *puntero a puntero* como base para manejar una celda desde la anterior. Esta solución, si bien no es muy habitual, es un buen ejemplo para ejercitarse los conocimientos que tenemos sobre punteros.

En la figura 6.4 se muestra un esquema gráfico de cómo vamos a controlar la posición de cada elemento. Como puede ver, tenemos una lista de 4 elementos controlada por el puntero *L*. Hemos dibujado dos punteros —*p* y *q*— que nos permiten referir dos posiciones distintas de la lista. Observe que estos punteros apuntan al puntero del que cuelga la celda con el elemento referido.

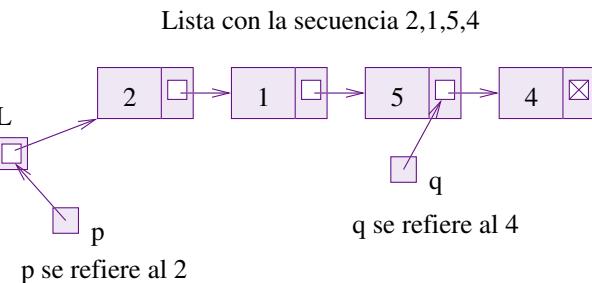


Figura 6.4

Cada celda está controlada por el puntero que la apunta.

Práctica: 6

Si una función que devuelve la posición de una celda, devuelve un puntero *al puntero del que cuelga la celda*, nos permitirá usar ese puntero para modificar directamente la celda correspondiente. Por ejemplo, podemos insertar una celda delante o incluso borrar dicha celda.

Tal vez le parezca un poco extraño que un puntero apunte al puntero que hay dentro de una celda. Sin embargo, no es más que un uso normal del operador `&`, aunque aplicado sobre un objeto miembro de una estructura. Por ejemplo, en la lista de la figura 6.4 podemos obtener los punteros:

```
Celda *p1, *p2;
p1= &L; // Puntero al puntero de la primera celda
p2= &(L->sig); // Puntero al puntero de la segunda celda
```

donde el primer puntero apunta al puntero original que controla la lista mientras que el segundo apunta a un campo *sig* de una celda.

<sup>1</sup>Aunque se sale de los objetivos del guión, es interesante apuntar que hay tipos de datos que incluso no admiten la asignación y que no permitirían realizar esta operación.

**Ejercicio 6.6 — Buscar puntero a puntero.** Añada una función *BuscarPuntero*, a la que se le pasa el puntero *L* y el dato a buscar y que devuelve un puntero al puntero del que cuelga dicho dato. Tenga en cuenta que si no lo encuentra, devolverá un puntero al campo siguiente de la última celda, es decir, a un puntero que es nulo.

Para probar la función, añada el código necesario en **main** para determinar si el dato anteriormente introducido está en la lista. Un ejemplo de ejecución es el siguiente:

```

prompt> ./celdas 15 9
Lista: {7,6,9,3,6,7,7,6,5,8,8,9,5,3,5}
La lista contiene 15 elementos.
Introduzca un dato a buscar: 9
Buscar celda: El dato está en la lista
Buscar puntero: El dato está en la lista.

```

Para implementarlo, deberá realizar una llamada a la función *BuscarPuntero*. En caso de que el puntero devuelto no apunte a un puntero nulo, el dato está en la lista. Observe en el ejemplo que el dato se consulta dos veces, pues ya teníamos una implementación que devolvía puntero a celda.

Cuando implemente la función, es importante tener en cuenta que si pasamos el puntero de la lista, éste pasa por referencia, aunque la lista no vaya a ser modificada, ¿Por qué?

*Nota: la tarea aparece en el listado como //FIXME 6.*

La devolución de un puntero a puntero nos permite realizar fácilmente el borrado del elemento en esa posición de la lista. Por ejemplo, se puede crear una función que simplemente descuelgue la celda de la lista y devuelva como resultado un puntero a la celda extraída. Si nuestra intención es borrarla, podemos hacer un **delete** a ese puntero para liberar la memoria.

**Ejercicio 6.7 — Eliminar todas las ocurrencias de un elemento.** Añada una función *Descolgar* que recibe un puntero a puntero a celda, descuelga la celda de la lista y la devuelve como resultado. Para comprobar que funciona como se espera, añada un trozo de código en **main** para eliminar todas las ocurrencias del elemento que acabamos de buscar. Para ello, tendrá que añadir un bucle que elimine el elemento que localiza la función *BuscarPuntero* hasta que no haya más repeticiones. Un ejemplo de ejecución es el siguiente:

```

prompt> ./celdas 15 9
Lista: {7,8,7,4,6,8,5,7,2,1,8,7,7,4,1}
La lista contiene 15 elementos.
Introduzca un dato a buscar: 7
Buscar celda: El dato está en la lista
Buscar puntero: El dato está en la lista.
Sin ese dato: {8,4,6,8,5,2,1,8,4,1}

```

Tenga en cuenta que para buscar la siguiente ocurrencia del elemento, no es necesario pasar el inicio de la lista a *BuscarPuntero*, sino la posición donde paró la anterior búsqueda. Además, no olvide que una vez devuelta la celda que hemos descolgado, habrá que liberarla.

*Nota: la tarea aparece en el listado como //FIXME 7.*

### 6.2.3 Ordenación

Los algoritmos de ordenación con celdas enlazadas de esta sección se presentan como algoritmos que reordenan las celdas, es decir, no movemos datos de una celda a otra, sino que reorganizamos los punteros para que queden ordenados. Esta forma de solucionarlo no sólo es la más habitual, sino la más interesante desde un punto de vista práctico.

Antes de entrar en los detalles de la ordenación de celdas enlazadas es importante recordar la naturaleza de esta estructura y la dificultad o poca eficiencia de una operación que intente acceder a la posición i-ésima. Teniendo esto en cuenta, es lógico deducir que cualquier algoritmo de ordenación que se exprese fácilmente con operaciones que recorren los datos de forma secuencial puede adaptarse fácilmente a una lista de celdas enlazadas. Por otro lado, un algoritmo que requiere *acceso aleatorio*, es decir, que acceden a distintos elementos en posiciones distantes —podríamos decir, “dando saltos”— no se podrá adaptar fácilmente o no tiene sentido en una implementación basada en celdas enlazadas.

Para comenzar esta sección, vamos a trabajar con un nuevo archivo **ordenar\_celdas.cpp** que habrá descargado con este guión. Para comenzar, vamos a aprovechar algunos ejercicios que ha resuelto en las secciones anteriores.

**Ejercicio 6.8 — Módulo de utilidades con celdas.** Considere el archivo `celdas.cpp` que ha obtenido como resultado de los ejercicios anteriores. Escriba un nuevo módulo C++ que contenga las funciones que ha resuelto y añadido en ese programa. Para ello, tendrá que crear dos ficheros:

- `util_celdas.h`: Un fichero cabecera que contendrá la definición de la estructura `Celda` y las cabeceras de funciones.
- `util_celdas.cpp`: Un fichero de implementaciones que incluirá el anterior y contendrá las definiciones de todas las funciones.

Los programas que ejecutará en esta sección serán el resultado de compilar y enlazar el código `util_celdas.cpp` con `ordenar_celdas.cpp`.

*Nota: la tarea aparece en el listado como //FIXME 1.*

## Algoritmo de selección

En primer lugar presentamos un algoritmo simple que reescribimos para ordenar una lista de celdas enlazadas: algoritmo de *ordenación por selección*. Recordemos que este algoritmo resuelto para vectores consiste en buscar la posición donde está el menor elemento e intercambiárselo con el primero; del resto de elementos, buscar el más pequeño y ponerlo el segundo, etc. El proceso se repite hasta que llegamos a la última posición.

Podemos reescribir el mismo algoritmo para el caso de celdas enlazadas. Para ello, podemos buscar la celda con el elemento más pequeño para ponerla en primer lugar y repetir el proceso hasta que la lista quede ordenada. En lugar de hacerlo así, vamos a crear una implementación en la que buscamos el elemento más grande para insertarlo en primer lugar en una nueva lista que comienza como vacía. Recuerde que añadir un elemento a una lista es mucho más fácil si se hace como primera celda, pues sólo tenemos que cambiar el puntero que controla la lista.

**Ejercicio 6.9 — Ordenar celdas por selección.** Implementar el algoritmo de ordenación por selección y usarlo para ordenar las dos listas del programa `ordenar_celdas.cpp`. Para ello, deberá incluir:

- Una función `BuscarMaximo` similar a la función `BuscarPuntero` ya que devuelve un puntero al puntero del que cuelga la celda con el valor máximo. Recibe como parámetro la lista y devuelve un puntero a puntero a celda.
- Una función `OrdenarSeleccion` que recibe una lista por referencia y la ordena con el algoritmo de selección.

Para probarlo, añade un trozo de código en `main` para ordenar las dos listas y listarlas ordenadas. Un ejemplo de ejecución es el siguiente:

```
prompt> ./ordenar_celdas 15 9
Lista1: {5,5,6,5,6,5,7,7,8,9,1,8,7,1,2}
Lista2: {6,8,5,2,6,2,1,9,3,7,8,9,7,7,6}
Lista1 ordenada: {1,1,2,5,5,5,6,6,7,7,7,8,8,9}
Lista2 ordenada: {1,2,2,3,5,6,6,6,7,7,7,8,8,9,9}
```

Tenga en cuenta que el algoritmo de ordenación deberá usar la primera función para encontrar la celda, la tendrá que descolgar —recuerde la función `Descolgar` ya resuelta— y la insertará en la primera posición de una nueva lista vacía. Este proceso se repite hasta que la lista original queda vacía.

*Nota: la tarea aparece en el listado como //FIXME 2.*

En el caso de la implementación anterior, es interesante indicar que la selección del elemento más grande debería hacerse de forma que la celda seleccionada sea la más alejada del comienzo. Es decir, que si encontramos el elemento mayor repetido, deberíamos seleccionar el más cercano al final. En principio parece una detalle irrelevante, pero recuerde que una característica deseable de un algoritmo de ordenación es que sea *estable*. El objetivo de ese detalle en la implementación es hacerlo estable, en contraposición a la versión basada en vectores para la que el resultado es un algoritmo inestable<sup>2</sup>.

## Mezcla de listas ordenadas

La mezcla de dos listas ordenadas se puede implementar muy eficientemente si creamos una nueva lista que rellenamos con todos los elementos que obtenemos descolgando las celdas de las dos listas originales. El algoritmo consiste en comparar repetidamente los dos primeros elementos de las listas, descolgar el más pequeño, e insertarlo al final de la nueva lista.

Note que el algoritmo que indicamos dice explícitamente que se insertará al final de la nueva lista. Si lo insertáramos siempre al principio, obtendríamos una lista ordenada pero con el orden cambiado, es decir, en nuestro caso de mayor a menor. Si queremos respetar el mismo orden, tendríamos que dar la vuelta a la lista.

**Ejercicio 6.10 — Mezclar celdas ordenadas.** Implemente un algoritmo de mezcla de listas ordenadas. Para ello, escriba una función `Mezclar` que recibe dos parámetros puntero a celda por referencia —las listas ordenadas— y

<sup>2</sup>Podríamos pensar son algoritmos distintos ya que tienen distintas características. Realmente, se basan en la misma idea, aunque la implementación provoca ese comportamiento diferenciado. Por ejemplo, podemos implementar *selección* en vectores copiando los mínimos en un nuevo vector y generando una versión estable. Lógicamente, prescindir de memoria extra requiere el intercambio de elementos en el mismo vector, lo que provoca la inestabilidad.

devuelve un puntero a la nueva lista mezcla. Note que no se realizará ninguna reserva ni liberación de celdas, ya que la nueva lista contendrá las mismas celdas originales. Por tanto, las dos listas de entrada quedarán vacías.

En la implementación tenga cuidado con insertar el siguiente elemento al final de la lista que está creando. Para hacerlo, deberá mantener un puntero que apunte a la última celda para no tener que buscar el final de la lista cada vez que quiera añadir una nueva celda.

Para probarlo, añada un trozo de código en `main` para mezclar las dos listas ordenadas y listar el resultado. Un ejemplo de ejecución es el siguiente:

```
Consola
prompt> ./ordenar_celdas 15 9
Lista1: {6,5,2,8,7,4,5,4,2,3,9,7,2,7,3}
Lista2: {8,9,5,5,9,1,9,9,8,1,2,3,8,2,1}
List1 ordenada: {2,2,2,3,3,4,4,5,5,6,7,7,8,9}
List2 ordenada: {1,1,1,2,2,3,5,5,8,8,8,9,9,9,9}
Lista mezcla: {1,1,1,2,2,2,2,3,3,4,4,5,5,5,6,7,7,7,8,8,8,9,9,9,9,9}
```

Tenga en cuenta que el algoritmo de mezcla deja las dos listas vacías, por lo que puede aprovechar el identificador `list1` para guardar la nueva lista ordenada y dejar `list2` como vacía.

*Nota: la tarea aparece en el listado como //FIXME 3.*

## Ordenación por mezcla

Un algoritmo especialmente eficiente y fácil de adaptar a la ordenación de celdas enlazadas es la ordenación por mezcla. La idea es muy sencilla y se puede expresar mediante un algoritmo recursivo: la ordenación de  $n$  elementos se puede realizar:

1. Ordenando los  $n/2$  primeros elementos.
2. Ordenando los  $n-n/2$  últimos elementos<sup>3</sup>.
3. Mezclando las dos secuencias ordenadas anteriores.

Lógicamente, los dos primeros pasos corresponden a aplicar el mismo algoritmo a una secuencia más pequeña. La recursividad se detiene cuando llegamos al caso de ordenar un único elemento, donde no hay que hacer nada, pues ya está ordenado. Es decir, el caso base es cuando la lista tiene tamaño 1.

La implementación en su programa `ordenar_celdas.cpp` se puede hacer fácilmente al disponer de la función `Mezclar`. Sólo tiene que realizar la división de la lista a ordenar en dos listas, ordenar recursivamente y mezclar el resultado para obtener la lista final.

**Ejercicio 6.11 — Ordenación por mezcla.** Escriba una función `MergeSort` que reciba una lista de celdas enlazadas por referencia y la ordene usando el algoritmo de ordenación por mezcla. Para probarla, modifique el código de la función `main` para obtener la primera lista ordenada con este algoritmo. Un ejemplo de ejecución es el siguiente:

```
Consola
prompt> ./ordenar_celdas 15 9
Lista1: {5,8,8,8,3,2,5,4,7,8,8,6,8,7,1}
Lista2: {4,1,6,8,5,6,8,1,7,5,9,2,6,3,3}
List1 ordenada (mergesort): {1,2,3,4,5,5,6,7,7,8,8,8,8,8}
List2 ordenada (selección): {1,1,2,3,3,4,5,5,6,6,6,7,8,8,9}
Lista mezcla: {1,1,1,2,2,3,3,3,4,4,5,5,5,6,6,6,7,7,7,8,8,8,8,8,9}
```

Tenga en cuenta que al dividir la lista de celdas enlazadas en dos partes, tendrá que avanzar un puntero desde el principio hasta parar en la celda anterior a la que quiere descolgar. Recuerde que tendrá que modificar el campo `sig` para descolgar la segunda `sublista` y hacer ese campo `sig` nulo para que las dos listas queden como listas independientes.

*Nota: la tarea aparece en el listado como //FIXME 4.*

## Analizando código

Como último algoritmo de ordenación, vamos a proponer un ejercicio de análisis de código ya resuelto. Para ello, tendrá que usar la siguiente función en su programa:

```
void OrdenarEspecial(Celda*& l)
{
    Celda* vec[32] = {0};
    Celda** tope= &(vec[0]);

    while (l) {
        Celda* acarreo= l;
        l= l->sig;
```

<sup>3</sup>También podría calcularse como  $(n+1)/2$ .

```

acarreo->sig= 0;
Celda**aux= &(vec[0]);

while (*aux != 0) {
    acarreo= Mezclar(acarreo,*aux);
    aux++;
}
*aux= acarreo;
if (aux==tope)
    ++tope;
}

for (Celda** p=&(vec[0]); p!=tope; ++p) {
    l= Mezclar(l,*p);
}
}

```

Está función —que ya tiene en el archivo que descargó— realiza una ordenación de la lista que se le pasa como parámetro. Para ello, usa como función auxiliar la función *Mezclar* que ya ha resuelto y probado.

**Ejercicio 6.12 — Analizar función.** Modifique su programa en la función **main** para que llame a esta función ya resuelta. Para ello, cambie la llamada a la ordenación por selección con una llamada a esta nueva función. Un ejemplo de ejecución es el siguiente:

```

Consola
prompt> ./ordenar_celdas 15 9
Lista1: {7,8,7,2,9,2,7,6,3,2,8,1,2,4,3}
Lista2: {7,7,3,4,9,5,3,9,5,8,6,5,9,4,2}
Lista1 ordenada (mergesort): {1,2,2,2,2,3,3,4,6,7,7,7,8,8,9}
Lista2 ordenada (especial): {2,3,3,4,4,5,5,5,6,7,7,8,9,9,9}
Lista mezcla: {1,2,2,2,2,2,3,3,3,3,4,4,4,5,5,5,6,6,7,7,7,7,8,8,8,9,9,9,9}

```

Una vez comprobado el correcto funcionamiento de esta nueva propuesta, analice la tarea que realiza la función —poniendo atención en ese vector de tamaño 32— y responda a la siguiente pregunta: ¿Cuál es el tamaño máximo de lista que puede ordenar?

*Nota: la tarea aparece en el listado como //FIXME 5.*

## 6.2.4 Rangos de elementos

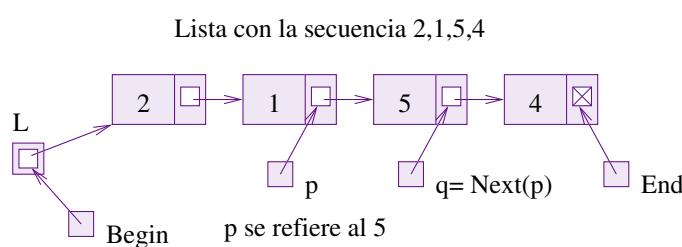
El concepto de rango de elementos se ha introducido en la sección 5.2.4 (página 58) como una secuencia de elementos en un contenedor delimitada por dos posiciones, la primera indicando el primer elemento y la segunda posición indicando el elemento siguiente al último.

Una lista de celdas enlazadas es un contenedor que puede usarse para ilustrar el concepto de rango de elementos. En esta sección presentamos algunos ejercicios con punteros y celdas a fin de que mejoren la formación sobre este tema, pero que también permitan al estudiante crear asociaciones de ideas entre estructuras muy diferentes, de forma que se fomente la reflexión sobre la abstracción de conceptos como posición y rango de elementos. En temas más avanzados, se volverá a estudiar y terminará siendo un pilar importante cuando conozca en profundidad el lenguaje.

En este tema proponemos usar la idea de puntero a puntero como controlador de una posición en la lista. Recuerde la función *BuscarPuntero* que implementó en el ejercicio 6.6. Esta función devuelve la posición donde se encuentra el elemento. Si generalizamos esta idea, descubrimos que en una lista de  $n$  elementos podemos hablar de  $n + 1$  posiciones: desde la posición del primer elemento hasta la que hay detrás del último.

Si piensa en los punteros desde los que cuelgan celdas encontrará que existe un primer puntero —el mismo *L*— del que cuelga la primera celda hasta un último puntero —que contiene cero— que correspondería a la celda detrás de la última.

En la figura 6.5 se presenta un ejemplo para que vea el sentido de la primera posición, de la posición detrás de la última y de la posición siguiente a una dada.



**Figura 6.5**

Cada celda está controlada por el puntero que la apunta.

**Ejercicio 6.13 — Funciones básicas de rangos de celdas.** Implemente tres funciones: *Begin*, *End*, *Next* que resuelvan el problema de obtener la posición del primer elemento, del elemento después de último y del siguiente elemento (véase la figura 6.5).

Observe que las dos primeras recibirán una lista por referencia y devolverán un puntero a puntero que corresponden a la posición del primer elemento y del siguiente al último. Lógicamente, si la lista está vacía, ambas coinciden, pues son un puntero a *L* que vale cero. Por otro lado, la función *Next* recibe un puntero a puntero —que no valdrá *End*— para devolver la posición del siguiente elemento. Note que la implementación de *End* puede usar *Next*.

*Nota:* la tarea aparece en el listado como //FIXME 8.

Para probar el comportamiento de las posiciones controladas por punteros a punteros a celdas podemos realizar un algoritmo simple, como es el de listar los elementos de una lista. Para ello, puede crear una función que imprime en la salida estándar todos los elementos de un rango, es decir, puede obtener cualquier rango o subsecuencia de la lista y llamarla para el caso concreto en que las posiciones sean *Begin* y *End*.

**Ejercicio 6.14 — Ejemplo de uso de rangos de celdas.** Añada una función *Imprimir* que recibe dos posiciones que determinan un rango. La función imprime todos los elementos del rango, es decir, desde la primera posición, sin llegar a imprimir la segunda, ya que corresponde a la siguiente al último elemento del rango. Para probarlo, modifique *main* para que liste de nuevo la última lista usando la nueva función. Un ejemplo de ejecución puede ser el siguiente:

### Consola

```
prompt> ./celdas 15 9
Lista: {8,4,3,6,8,3,6,2,6,1,1,4,4,8,4}
La lista contiene 15 elementos.
Introduzca un dato a buscar: 4
Buscar celda: El dato está en la lista
Buscar puntero: El dato está en la lista.
Sin ese dato: {8,3,6,8,3,6,2,6,1,1,8}
Listado del rango total: {8,3,6,8,3,6,2,6,1,1,8}
```

Observe que el formato de la función *Imprimir* es el mismo que el de *Listar*. Note que tendrá que llamar a la función *Imprimir* con las posiciones *Begin* y *End*.

*Nota:* la tarea aparece en el listado como //FIXME 9.

Por último, es importante indicar que esta interpretación de la posición en una lista enlazada no deja de ser una implementación concreta de esta idea. Cuando estudie temas de estructuras de datos lineales volverá a trabajar el problema y seguramente encontrará varias alternativas, incluso más interesantes que ésta. Finalmente, cuando trabaje con el tipo *list*<> de la *STL*, las listas serán doblemente enlazadas, por lo que la implementación cambiará, aunque el concepto de rango será el mismo.

# Conecta N

Introducción.....	73
El juego Conecta-4.	
Problema a resolver.....	74
Puntuación de las partidas.	
Ejemplo de ejecución	
Diseño propuesto: versión 1 .....	77
Interfaces e implementación	
Programa de la versión 1	
Modificación del programa: versión 2 .....	79
Cambios internos a un módulo	
Cambios en la interfaz de un módulo	
Ampliar la funcionalidad del programa	
Práctica a entregar .....	84

## 7.1 Introducción

En esta práctica se propone crear un programa para jugar a una versión generalizada del conocido juego *Conecta-4*. Dado que estamos en un curso básico donde queremos introducir los conceptos de encapsulamiento, crearemos una versión simplificada basada en la interacción con la consola.

El objetivo de este guión es introducir al estudiante en el uso de nuevos tipos de datos que encapsulan la representación para ofrecer una interfaz que aísle al resto de módulos de los detalles de implementación. Por tanto, los crearemos con **class** y usaremos las palabras clave **private** y **public** para diferenciar la parte interna de la interfaz pública.

Siendo un ejercicio para introducir el uso de clases, el problema se dividirá en dos partes. La primera se basará en crear tipos simples, sin incluir estructuras de datos complejas ni la necesidad de memoria dinámica. En la segunda, se incluirá la memoria dinámica y distintos ejercicios de modificación y ampliación de la primera con el objetivo de mostrar los efectos del encapsulado que se propone, así como de las ventajas de esta metodología en un desarrollo más cercano a la realidad.

La solución del guión estará basada en el desarrollo de clases que se basan en tipos básicos del lenguaje, incluyendo punteros, *vectores-C*, *cadenas-C* y memoria dinámica. Por tanto, deberá evitar el uso de tipos de la *STL*.

### 7.1.1 El juego Conecta-4.

El conocido *Conecta-4* consiste en un tablero —de 7 columnas por 6 filas— colocado de forma vertical, donde se insertan hasta 42 fichas circulares, 21 de un color y 21 de otro. El juego consiste en que dos jugadores van colocando fichas de forma alternativa sobre el tablero, hasta que uno de los dos consigue situar 4 en línea, es decir, 4 fichas consecutivas en horizontal, vertical o diagonal.

Añadir una nueva ficha se realiza seleccionando una de las 7 columnas e introduciendo la ficha por ésta para que caiga hasta la posición libre más baja de dicha columna. Es decir, el tablero se va llenando desde abajo hacia arriba. En el caso de que ninguno de los jugadores obtenga 4 fichas en línea y no queden más posiciones, se considera empate.

En la figura 7.1 se presenta un ejemplo de un tablero donde se han insertado 16 fichas, con colores rojo y negro. Observe que el jugador con las fichas de color rojo ha ganado la partida, ya que ha conseguido poner 4 en línea, en este caso, en diagonal.

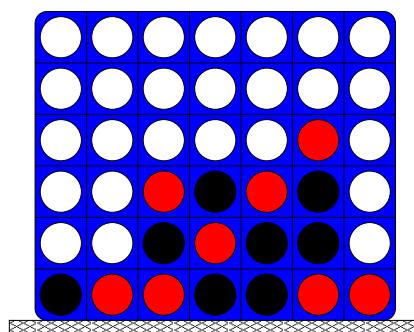


Figura 7.1  
Tablero de Conecta-4.

## 7.2 Problema a resolver

En este guión práctico vamos a crear un programa que nos permite jugar al juego *Conecta-N*. La interfaz se va a diseñar para la consola, mostrando un tablero sólo con caracteres. Por ejemplo, a continuación mostramos una situación donde se dibuja un tablero de  $6 \times 7$  con algunas fichas ya insertadas:



```
a b c d e f g
| | | | | | | |
| | | | | | |
| | | | | | |
|x| | |x| | | |
|o| | |o|x|o| |
|x|o|o|x|x|x|o|
=====
```

El juego que se propone generaliza el clásico juego de *Conecta-4* (sección 7.1.1). En lugar de diseñarse para un tablero de  $6 \times 7$  con el objetivo de conectar 4 fichas en línea, se programará para un tamaño de tablero y número de fichas objetivo variables. Básicamente, deberá controlar dos jugadores y un tablero para jugar tantas partidas como se desee, gestionando tanto el funcionamiento del tablero como las puntuaciones de los jugadores.

### 7.2.1 Puntuación de las partidas.

El programa debe establecer un criterio de puntuación cuando un jugador gana una partida. En principio, el método más simple para puntuar consiste en añadir un punto por cada partida ganada. De esta forma, el valor final de varias partidas entre dos contrincantes corresponde al número de partidas que se han ganado. Sin embargo, puede resultar más interesante que la puntuación varíe dependiendo de la partida, ya que no todos los éxitos tienen el mismo mérito. Por ejemplo, un jugador que gane en pocos turnos debería tener mejor puntuación que otro que gane en muchos.

El estudiante deberá implementar una función de puntuación variable dependiendo de la partida. Por ejemplo, algunos criterios que pueden influir en el valor pueden ser:

- *Tamaño del tablero*. Un tamaño de tablero mayor puede implicar mayor dificultad, y por tanto, mayor puntuación si se gana.
- *Número de fichas jugadas*. En un determinado tablero, ganar con un número de pasos menor podría implicar una mejor puntuación. Si no se desea que el tamaño del tablero afecte a la puntuación, se podría usar la proporción de posiciones ocupadas (o vacías).
- *Longitud de la línea*. Parece razonable que si se consigue un *6-en-línea*, se obtenga una puntuación superior que con un *4-en-línea*.

Por consiguiente, cuando un jugador gana una partida, podemos añadir un número de puntos que dependa de algunos de estos criterios o de otros que el alumno considere oportunos. Por ejemplo, podemos definir una función proporcional al número de posiciones libres del tablero y a la longitud de la línea.

El alumno debe seleccionar la función de puntuación que crea más conveniente e implementarla en la solución que proponga.

### 7.2.2 Ejemplo de ejecución

Para ejecutar el programa, será necesario conocer la configuración básica: dimensiones del tablero, número de fichas a alinear y nombres —o apodos— de los jugadores. El comienzo podría ser como sigue:



```
prompt> ./conectaN
Introduzca filas: 6
Introduzca columnas: 7
Introduzca fichas a alinear (3-5): 4
Introduzca nombre del primer jugador: Fulano
Introduzca nombre del segundo jugador: Mengana
```

Observe que en este ejemplo el número de fichas a alinear está entre 3 y 5. En la práctica, obligaremos a que el número de filas y columnas esté entre 4 y 20. Una vez conocido el tamaño, el número de fichas será un valor desde 3 al mínimo de los anteriores menos uno. Esto garantiza que pueden darse alineaciones verticales, horizontales y diagonales.

Después de conocer el tablero, se preguntan los nombres de cada jugador para poder referirse a cada uno de ellos a lo largo de la partida. Una vez introducidos estos datos, se comienza una partida en la que se van dando turnos a cada uno de los jugadores hasta que la partida termina, ya sea con un empate o con uno de los jugadores como ganador. La inserción de fichas

se hará indicando en la consola la letra que corresponde a la columna seleccionada. Por ejemplo, tras introducir los datos podemos obtener el siguiente tablero:

```
Consola
a b c d e f g
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
=====
Turno: jugador 1: (x)
Fulano, escoja una columna (letra a-g):
```

Si escogemos la columna '**a**', se inserta una ficha representada por el carácter '**x**' de la siguiente forma:

```
Consola
Turno: jugador 1: (x)
Fulano, escoja una columna (letra a-g): a

a b c d e f g
| | | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| |x| | | | | |
=====
Turno: jugador 2: (o)
Mengana, escoja una columna (letra a-g):
```

Una nueva inserción —del segundo jugador— en la misma columna dará como resultado:

```
Consola
Mengana, escoja una columna (letra a-g): a

a b c d e f g
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
|o| | | | | |
|x| | | | | |
=====
Turno: jugador 1: (x)
Fulano, escoja una columna (letra a-g):
```

El juego continúa hasta que haya una inserción que dé lugar a una línea o se haya completado el tablero sin que haya ganador. Por ejemplo, una simulación muy simple para terminar la partida podría llegar hasta:

**Consola**

```

Turno: jugador 1: (x)
Fulano, escoja una columna (letra a-g): d

a b c d e f g
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
|o|o|o| | | |
|x|x|x|x| | |
=====

Partida finalizada. Ganador: jugador 1
Resultados tras esta partida:
  Fulano: 1 ganadas que acumulan 77 puntos
  Mengana: 0 ganadas que acumulan 0 puntos
¿Jugar de nuevo(S/N)?:

```

Como puede ver, se presentarán los resultados hasta este momento y se ofrecerá la posibilidad de jugar de nuevo. En caso de que se responda con la letra '**s**', volverá a presentar la situación inicial:

**Consola**

```

¿Jugar de nuevo(S/N)?: s

a b c d e f g
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
=====

Turno: jugador 1: (x)
Fulano, escoja una columna (letra a-g):

```

Suponga que se repite la misma partida y finalmente se indica que no se quiere seguir. El final de la partida podría ser el siguiente:

**Consola**

```

a b c d e f g
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
|o|o|o| | | |
|x|x|x|x| | |
=====

Partida finalizada. Ganador: jugador 1
Resultados tras esta partida:
  Fulano: 2 ganadas que acumulan 154 puntos
  Mengana: 0 ganadas que acumulan 0 puntos
¿Jugar de nuevo(S/N)?: n
Resultados finales:
  Fulano: 2 ganadas que acumulan 154 puntos
  Mengana: 0 ganadas que acumulan 0 puntos
  0 empatadas

```

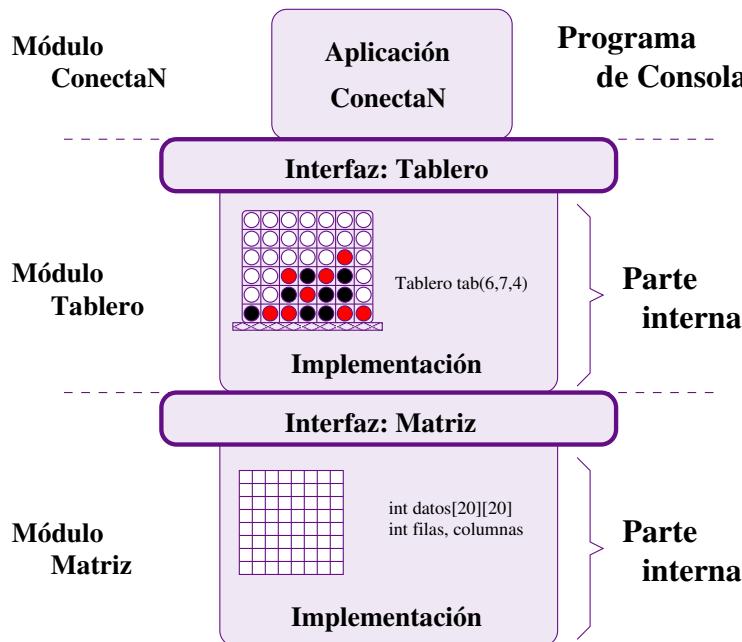
Observe que el primer jugador a acumulado la misma cantidad de puntos que antes y en los resultados finales aparece el número de partidas empatadas.

## 7.3 Diseño propuesto: versión 1

El objetivo es mostrar la facilidad de generar una solución fácilmente mantenible en base a la modularización y el diseño de clases. En primer lugar, proponemos la creación de una clase *Tablero* que se encargará de almacenar y gestionar el funcionamiento de un tablero.

Si bien podría ser una solución suficiente, enfatizaremos la capacidad de encapsulamiento de una clase dividiendo el problema en dos clases: la clase tablero propiamente dicha y la clase *Matriz* que usaremos para encapsular la representación de una estructura bidimensional.

Con estas dos clases, nuestro programa se puede esquematizar como tres módulos independientes que permiten encapsular la representación mediante la interfaz de una clase. En la figura 7.2 puede observar que nuestro programa trabajará con la interfaz del tablero, sin conocer nada sobre su implementación interna.



**Figura 7.2**

Independencia entre el módulo principal y la implementación matriz.

Además de estas dos clases, podemos añadir la clase *Jugador* que de forma que nuestro programa tendrá que implementarse en base a la interacción de dos jugadores y un tablero. Por tanto, el diseño propuesto se basa en:

- Clase *Tablero*. Un objeto de esta clase almacenará y gestionará el funcionamiento de un tablero.
- Clase *Matriz*. Encapsula la representación de una matriz de enteros, usada para representar un tablero. Con ella, podremos implementar un tablero sin necesidad de conocer los detalles de la estructura de datos que mantiene el contenido del tablero.
- Clase *Jugador*. Un objeto de esta clase mantendrá los datos relacionados con un jugador y nos facilitará la interacción con el tablero.

Una representación más completa del juego se puede ver en la figura 7.3. Observe que en esta figura hemos incluido el cuarto módulo: *Jugador*. Este módulo usará el tablero, al igual que la aplicación *conectaN*. De nuevo, la implementación del módulo *Matriz* es independiente del resto del programa.

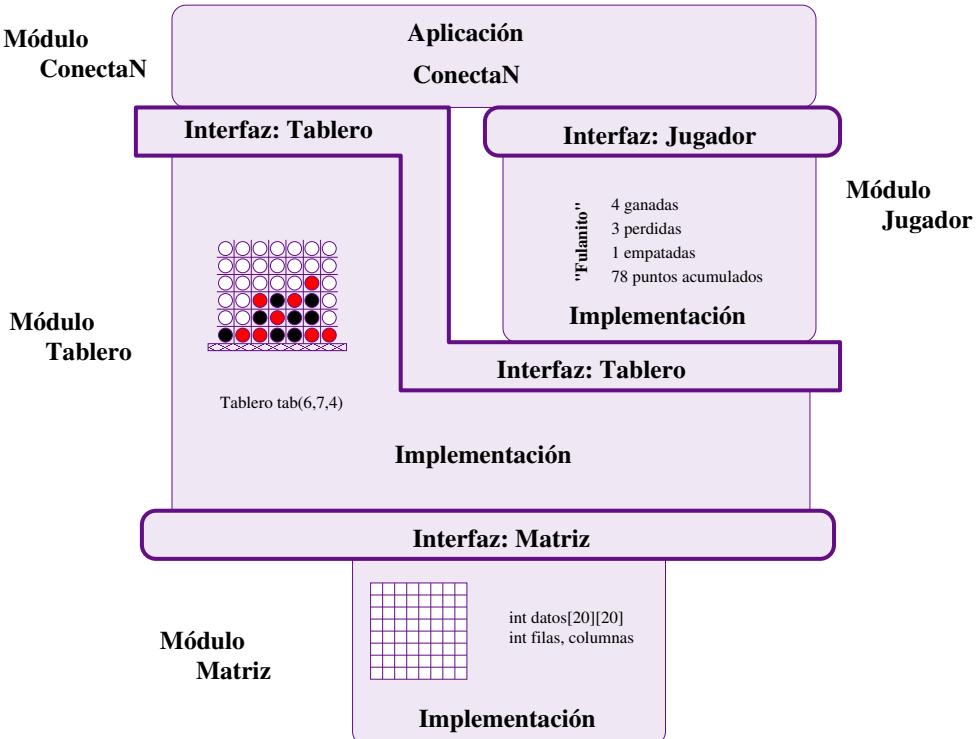
Una perspectiva muy interesante de la figura 7.3 es ver cómo todo el programa se diseña sobre la estructura de datos que implementa la representación del tablero. Lógicamente, en este juego vamos a modificar y gestionar los cambios del tablero. Estos cambios afectarán a la matriz que representa el tablero. Observe que si decidimos modificar la implementación de la clase *Matriz*, todo el programa seguirá siendo válido siempre que este cambio no afecte a la interfaz del módulo.

### 7.3.1 Interfaces e implementación

La mejor forma de entender los detalles del diseño propuesto es especificar con más detalle cuáles son las representaciones y las interfaces que se proponen. Concretamente, daremos algunas indicaciones para implementar las clases de la figura 7.3: *Matriz*, *Tablero* y *Jugador*.

#### Clase Matriz

El objetivo de esta clase es encapsular la estructura de datos que mantiene la matriz que representa el tablero. Se propone usar como parte privada una *matriz-C* fija de tamaño 20 por 20. Con esta matriz, podremos representar cualquier tablero que tenga hasta 20 casillas para cada dimensión. Cada posición de la matriz contendrá un valor entero con valores 0, 1 y 2, que indican vacío, ficha de color 1 y ficha de color 2, respectivamente.



**Figura 7.3**  
Cuatro módulos para implementar el juego *Conecta-N*.

Respecto a las operaciones que ofrece esta clase en la interfaz, se podrían incluir las siguientes:

- Consulta del número de filas del tablero.
- Consulta del número de columnas del tablero.
- Consulta del elemento en una posición determinada.
- Modificación del valor de una determinada posición.

Por lo tanto, el resultado es un contenedor de enteros organizados en una estructura bidimensional. Esta interfaz nos permite trabajar con la clase sin pensar en la forma en que se almacenan los valores internamente.

Note que el hecho de haber escogido una *matriz-C* de tamaño 20 por 20 es relevante, porque implica que con un tablero de 6 por 7 vamos a usar sólo 42 de las 400 posibles posiciones. Sin embargo, esta decisión no es especialmente comprometedora, porque es fácil modificarla —como veremos más adelante— si no cambiamos la interfaz.

### Clase Tablero

La clase tablero implementa el comportamiento de un tablero colocado en posición vertical y en el que se insertan fichas en cada columna hasta el fin del juego. El tablero se crea a partir de sus dimensiones y el número de fichas que hay que alinear. Por simplicidad, lo podemos definir como un contenedor de enteros, donde el número cero codifique posición vacía, el número 1 ficha del primer jugador y el número 2 ficha del segundo jugador.

Para entender mejor esta clase, listamos algunas de las operaciones que podría ofrecer:

- Operaciones de consulta:
  - Consulta de número de filas y columnas.
  - Consulta del número de fichas objetivo a alinear.
  - Consulta del contenido de una posición del tablero.
  - Consulta si la partida está finalizada.
  - Consulta el turno actual. No sólo mantiene el tablero, sino también cualquier información relacionada sobre el estado de la partida. Por ejemplo, el turno actual.
  - Consulta quién es el ganador. Puede devolver 0 en caso de empate o que no esté finalizada.
  - Consulta la puntuación obtenida en la partida actual. Como precondición, la partida debe estar finalizada.
- Operaciones que modifican el tablero:
  - Insertar en una columna. Note que no es necesario indicar quién inserta. El tablero es el encargado de gestionar los turnos, por lo que la columna es el único dato necesario para la inserción.
  - Vaciar el tablero. Vacía el tablero, es decir, lo reinicializa vacío con las mismas dimensiones. Nos permite volver a comenzar una nueva partida.

Además, podemos añadir alguna operación más si lo considera conveniente. Por ejemplo, puede añadir la operación *prettyPrint* que imprime el tablero en la salida estándar para interactuar con el usuario. Se puede usar el formato que

hemos visto en los ejemplos anteriores, con el tablero seguido de un mensaje que indica si se ha finalizado o el turno que corresponde a la siguiente inserción.

### Clase Jugador

La tercera clase de la solución es la clase *Jugador*. Dos objetos de esta clase interaccionarán con un objeto de la clase *Tablero* para implementar el desarrollo de una partida de *cuentaN*.

Esta clase tiene como objetivo mantener los datos relacionadas a cada jugador; básicamente, el nombre, el turno que le corresponde y las puntuaciones obtenidas. Para facilitar el control de esta puntuación acumulada, puede añadir una función miembro que recibe un tablero de la partida finalizada y el jugador contabiliza el resultado.

Por otro lado, deberá implementar una función *escogeColumna* que realice la elección de una columna. Para nuestro programa, bastará con una función que imprime el tablero en la salida estándar, solicita una columna válida para insertar una ficha y finalmente modifica el tablero con dicha inserción. La función devuelve **void** y repite la pregunta de la columna mientras no sea correcta<sup>1</sup>.

La implementación deberá contener una cadena de caracteres con el nombre del jugador. Para esta primera versión, dado que no queremos usar estructuras complejas ni memoria dinámica, basta con declarar una cadena de hasta, por ejemplo, 50 caracteres. Al crear el jugador no será posible asignar un tamaño superior, por lo que se registrarán únicamente los caracteres que quiepan.

Un detalle que tal vez le resulte más complicado es la implementación de un método de consulta del nombre. Tal vez esté tentado a implementar una función que recibe un parámetro de tipo vector de caracteres para modificarlo con el nombre. Lo más simple y recomendado es que la función devuelva un puntero al nombre que se almacena. Lógicamente, será un puntero que no permite modificar el contenido del vector.

### 7.3.2 Programa de la versión 1

Las clases que se han desarrollado en las secciones anteriores se pueden unir junto con un módulo principal —que contendrá **main**— para obtener la funcionalidad que se ha expuesto en la sección 7.2 (página 74). Tenga en cuenta que jugar una partida a *cuentaN* no es más que iterar mientras que la partida no haya finalizado, de forma que en cada iteración se pregunta al tablero de quién es el turno y se le da la opción de insertar al jugador correspondiente.

En el desarrollo de este programa tendrá que integrar los módulos anteriores junto con el código que gestionará el desarrollo de las partidas. Conviene que los errores de integración sean mínimos y fáciles de localizar. Para conseguirlo, es aconsejable insertar código de ayuda en la depuración. Por ejemplo, puede insertar aserciones que garanticen que el estado del programa es correcto. Algunas condiciones útiles para las clases anteriores son:

- La modificación o consulta de una posición en la matriz debe incluir una posición válida. Es decir, deben ser valores que van del cero al tamaño menos uno (tanto para filas como para columnas).
- La inserción de una ficha en una columna sólo tiene sentido si hay espacio libre en la columna.
- Solicitar que un jugador escoja una columna sólo tiene sentido si el turno que controla el tablero coincide con el turno del jugador.
- Solicitar la puntuación que corresponde a un tablero no tiene sentido si la partida está sin finalizar.

Si en algún momento de la ejecución el programa llama a estas operaciones sin sentido, es lógico pensar que ha habido un error previo que ha llevado a ese estado. En ese momento, debería pararse el programa y resolver el problema antes de continuar.

**Ejercicio 7.1 — Programa versión 1.** Incluya las aserciones que se han indicado, cree el módulo principal e intégrelos para generar el programa que corresponde a la versión 1.

## 7.4 Modificación del programa: versión 2

Una vez resuelta la primera versión, se proponen una serie de modificaciones para generar un nuevo programa. En la práctica, las modificaciones de un programa vendrán principalmente determinadas por la corrección de errores, mejora de la eficiencia o la incorporación de nueva funcionalidad. La mayoría de las propuestas de estas sección tienen esa intención, aunque lo más importante es que entienda que las modificaciones pueden ser de muchos tipos y que un buen programa debería facilitar dichos cambios.

Se aconseja realizar los cambios en el orden en que se van presentando en las siguientes secciones, aunque el resultado final sea el mismo.

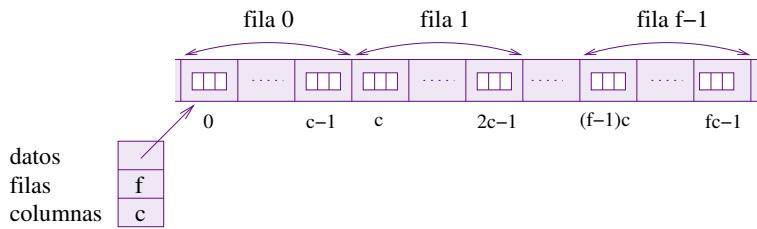
### 7.4.1 Cambios internos a un módulo

La primera modificación que vamos a realizar al programa consiste en cambiar sólo la parte interna de un módulo, dejando la interfaz intacta. La consecuencia de este cambio es que el resto del programa debería seguir siendo el mismo, ya que los cambios no le afectan.

<sup>1</sup>Se indican estos detalles porque más adelante se indicará una posible modificación.

## Cambio de la representación de la matriz

Sin entrar en una discusión sobre la mejor representación de esta clase, vamos a realizar un cambio en la representación para que no haya limitación en el tamaño. El cambio consiste en situar los valores de la matriz en un vector reservado en memoria dinámica. Es decir, tendremos un puntero que apunta a tantas posiciones como el producto de filas por columnas. En la figura 7.4 puede ver el tipo de representación que se propone.



**Figura 7.4**  
Representación de una matriz en memoria dinámica.

Como consecuencia de este cambio, las operaciones de copia y de asignación por defecto no son válidas, por lo que deberá implementarlas para completar la matriz. Una vez añadidas, junto con el destructor, el resto del programa deberá funcionar sin ningún problema.

**Ejercicio 7.2 — Cambiar la representación de la matriz.** Modifique la clase *Matriz* para que almacene los valores de cada posición en memoria dinámica. Recuerde que deberá incluir el constructor de copiar, operador de asignación y destructor. Si alguna de las dimensiones es cero, puede representar la matriz vacía con un puntero nulo.

## Cambio en la representación del jugador

La limitación de 50 posiciones en el tamaño de la cadena que guarda el nombre del jugador se puede eliminar si creamos ese espacio en memoria dinámica. Deberá modificar ese atributo para usar un puntero que apunta a la cadena que contiene el nombre.

Como consecuencia de este cambio, será necesario ampliar la clase para garantizar un correcto funcionamiento. En concreto, deberá añadir el constructor de copias, el operador de asignación y el destructor.

**Ejercicio 7.3 — Cambiar la representación de un jugador.** Modifique la clase *Jugador* para que pueda almacenar un nombre de cualquier longitud.

## 7.4.2 Cambios en la interfaz de un módulo

Los cambios en la interfaz son más complicados de realizar, ya que no sólo afecta al módulo que contiene dicha interfaz, sino a cualquier otro módulo que la haya usado. Por tanto, son cambios que deberían meditarse más detenidamente.

Cuando seleccionamos la implementación interna, es posible elegir soluciones simples para obtener un prototipo rápido pensando en que es fácil cambiarlo. Sin embargo, la interfaz debería seleccionarse pensando que un cambio será, como poco, más complicado y propenso a errores. En el peor de los casos, un cambio en la interfaz podría provocar la invalidación de múltiples programas que la hayan usado así como un costoso esfuerzo para volver a actualizarlos. Por ello, la interfaz debería diseñarse con más cuidado, pensando tanto en resolver el problema actual como en los futuros cambios.

### Compatibilidad hacia atrás

Uno de los problemas de cambiar la interfaz es que todos los programas que la hayan usado dejan de ser válidos. Es decir, no podrán compilarse con el nuevo módulo. Una forma de resolver este problema es modificar la interfaz pero garantizando que los programas desarrollados siguen siendo válidos. Realmente, la interfaz anterior sigue siendo válida, aunque ahora se amplía.

En el programa que ha desarrollado en las secciones anteriores, se propone cambiar la función que imprime el tablero formateado. Recuerde que se presentó como la función *prettyPrint* sin parámetros. El cambio consiste en reescribir la función con un parámetro: un flujo de salida (tipo *ostream*). Si cambiamos la función de esta forma, los programas que la usaban dejarán de ser compilables. Para resolverlo, cambiamos la implementación manteniendo la compatibilidad. Para ello tenemos dos posibilidades:

- Mantenemos dos funciones sobrecargadas. Recuerde que podemos usar el mismo nombre para dos funciones que difieren en los parámetros.
- Cambiar la función añadiendo el parámetro e incluir un valor por defecto.

En la práctica, la segunda opción es la mejor solución. Tenga en cuenta que si tenemos dos funciones, serían casi iguales. De hecho, un cambio en una probablemente también habría que hacerlo en la otra.

**Ejercicio 7.4 — Cambio de la interfaz con parámetro y valor por defecto.** Modifique la función `prettyPrint` añadiendo un parámetro de tipo `ostream` cuyo valor por defecto será `cout`.

Otra forma de cambiar la cabecera de una función sin que afecte a programas ya desarrollados es ampliar su funcionalidad devolviendo un error en lugar de `void`. Note que si antes devolvía `void`, nunca se asignó a ningún sitio. Si la nueva función devuelve un valor, los códigos anteriores podrían ser válidos si pueden ignorar el valor devuelto.

**Ejercicio 7.5 — Cambio de la interfaz añadiendo devolución.** Modifique la función `escogeColumna` de la clase `Jugador` para que devuelva un valor booleano que indica si ha tenido éxito. Si la columna no es correcta, no realiza inserción y devuelve que ha habido un error.

### Marcar como obsoleto

El problema es más complicado cuando queremos cambiar una interfaz y no es posible compatibilizarla con la anterior. En este caso, el cambio inevitablemente implica que programas ya terminados dejarán de ser válidos. Si las consecuencias del cambio son demasiado graves, podríamos incluso rechazar el cambio. Sin embargo, podemos optar por una solución intermedia: incluir nuevas posibilidades pero manteniendo la antigua interfaz.

La idea consiste en que queremos una nueva interfaz pero no queremos que los programas que usan la antigua dejen de ser válidos. En lugar de cambiarla y provocar la necesidad de cambiar el código antiguo, se avisa de que el cambio se llevará a cabo en el futuro.

El resultado es que hay dos formas de hacer las cosas. El nuevo módulo se publica, indicando qué partes de la interfaz se consideran obsoletas y qué partes se recomienda usar. Se usa la palabra *deprecated*, que podríamos traducir como *desaprobado* o *obsoleto*. Cualquier software que se desarrolle, debería evitar la funcionalidad marcada como *deprecated* pues en futuras versiones podrían directamente eliminarse<sup>2</sup>. Por otro lado, cualquier programa que queramos garantizar a largo plazo debería ser reconsiderado para adaptarse a la nueva interfaz.

### Ampliación de la funcionalidad de un módulo

Modificar un módulo para ampliar la funcionalidad es una operación que, en principio, no implica demasiados problemas. El único punto que podría ser delicado en esta ampliación es seleccionar una buena interfaz que garantice que será útil y perdurará en futuras revisiones.

Note que hablamos de ampliar la funcionalidad, dejando la actual intacta. Lógicamente, un cambio profundo podría dar lugar a rediseñar el módulo y cambiar la interfaz por completo; en este caso, no es tanto una ampliación sino un reemplazo del módulo.

Para mejorar la funcionalidad de los módulos de nuestro programa vamos a incorporar las operaciones de E/S que nos permitan salvar el estado del juego en un momento determinado. Para ello, será necesario:

1. Incluir operaciones de E/S como texto para el tipo `Matriz`. Para ello, podemos sobrecargar los operadores de E/S para los tipos `istream` y `ostream`.
2. Incluir operaciones de E/S como texto para el tipo `Tablero`. Estas operaciones almacenan las dimensiones del tablero seguidas por el contenido de cada una de las posiciones. Usará, por tanto, las operaciones propuestas en el punto anterior.
3. Incluir operaciones de E/S como texto para el tipo `Jugador`. En la sobrecarga de los operadores de E/S para este tipo de dato, deberá tener en cuenta que el formato de texto consiste en:
  - a) Una línea con el nombre del jugador. Puede suponer que la línea comienza con el carácter '`#`', seguida del nombre del jugador. De esta forma, podrá saltarse líneas vacías o que no corresponden al jugador.
  - b) Tras esta línea, puede suponer que los datos relacionados con el conteo del jugador están en formato de texto como enteros separados por "espacios blancos".

**Ejercicio 7.6 — Ampliar la funcionalidad.** Amplíe los tipos `Matriz`, `Tablero` y `Jugador` con la sobrecarga de los operadores de E/S.

### 7.4.3 Ampliar la funcionalidad del programa

En esta sección realizaremos una ampliación de la funcionalidad del programa aprovechando los cambios que se han propuesto en las secciones anteriores. El objetivo es entender que un diseño adecuado debería facilitar el mantenimiento, incluyendo la ampliación de la funcionalidad del programa.

#### E/S de la configuración y estado del juego

Las operaciones de E/S que se han implementado nos permiten resolver el problema de almacenar la configuración de una partida e incluso el estado intermedio de una partida. Para añadir esta funcionalidad, se diseña el formato de un fichero de configuración de partida con los siguientes bloques de información:

<sup>2</sup>Es interesante puntualizar que en C++14 ya se ha incorporado un atributo *deprecated* para marcar una entidad como obsoleta. Con ello, el mismo compilador generará el aviso para que consideremos otra implementación.

1. La primera línea es una cadena `#MP-CONECTAN-1.0` que distingue al fichero como configuración del juego *ConectaN*.
2. Los parámetros que definen al primer jugador. Note que deberán usarse las funciones de E/S del tipo *Jugador*, tanto para leerlo o salvarlo.
3. Los parámetros que definen al segundo jugador.
4. La configuración del tablero. De nuevo, se usarán las operaciones de E/S del tipo *Tablero*.

Para salvar el estado actual de una partida, podemos modificar el programa para que pueda leer y escribir este tipo de archivos. Para ello, puede incluir dos funciones *Cargar* y *Salvar* a las que se les pasa el nombre de un archivo así como objetos de tipo *Jugador* y *Tablero* para que realicen esas tareas. Si lo desea, puede incluirlas como funciones globales en el mismo archivo `cpp` donde tiene el `main`.

La forma de incorporarlo en la interfaz del programa puede ser aprovechando la lectura de columna cuando un jugador escoge dónde insertar. Recuerde que la función que se ha propuesto devuelve si ha tenido éxito. Por ejemplo, si solicitamos una columna y no introducimos una letra, la función devolverá que ha fallado. En este caso, podemos suponer que desea realizar otra operación: el salvado de la partida.

Un ejemplo de este tipo de ejecución es el siguiente:

```

Consola
Turno de jugador 1: (x)
Fulano, escoja una columna (letra a-g): a

a b c d e f g
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| x | | | | | |
=====

Turno de jugador 2: (o)
Mengana, escoja una columna (letra a-g): ?
Error en la selección de la columna. ¿Quiere guardar la partida? s
Introduzca nombre de archivo: inicio.conf
  
```

Observe que hemos introducido '`?`' para que falle la lectura de columna y dé la opción de guardar la partida. En este caso, se introduce un nombre de archivo como destino. Cuando se salva la partida, podría continuar por donde se dejó:

```

Consola
Turno de jugador 2: (o)
Mengana, escoja una columna (letra a-g): ?
Error en la selección de la columna. ¿Quiere guardar la partida? s
Introduzca nombre de archivo: inicio.conf

a b c d e f g
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| x | | | | | |
=====

Turno de jugador 2: (o)
Mengana, escoja una columna (letra a-g):
  
```

Por otro lado, para cargar una partida podemos hacerlo como forma de inicio del programa. Para ello, supondremos que el programa puede iniciarse de dos formas:

1. Llamada sin parámetros. Es decir, como hasta ahora; esta forma de lanzamiento realiza las operaciones de entrada manual de dimensiones y configuración de jugadores.
2. Llamada con un parámetro. Si la función `main` recibe un parámetro adicional, se entenderá que corresponde al nombre de archivo de configuración de la partida. En lugar de pedir manualmente los datos de la partida, se cargará el archivo de configuración y se continuará la partida por donde se quedó.

Por otro lado, también se puede incluir un último trozo de código al final del programa para salvar la partida actual. Después de la última partida, si se indica que no se quiere seguir jugando, se puede solicitar un posible archivo de configuración. El objetivo de esta opción es que los jugadores puedan continuar en el futuro con las puntuaciones conseguidas hasta ese instante.

**Ejercicio 7.7 — Cargar/Salvar partidas.** Modifique el programa para incorporar la funcionalidad que se ha indicado. Recuerde que tendrá que añadir dos funciones *Cargar* y *Salvar* al módulo principal y usarlas en el programa con la interfaz que se ha indicado. *Sugerencia: Puede plantear también una función DialogoSalvar donde encapsular el diálogo que se establece para salvar una partida y que se usa en más de un lugar.*

Finalmente, es interesante observar que las partidas siempre comienzan de la misma forma, es decir, con el tablero vacío y el jugador 1 con el turno. Si queremos dar la posibilidad de lanzar múltiples partidas consecutivas y acumular las puntuaciones en un enfrentamiento “largo”, deberíamos dar la opción de que el segundo jugador sea el que comienza. Si ha desarrollado los módulos como se ha descrito en el guión, tal vez le resulte fácil añadir esta posibilidad cambiando el módulo *Tablero*.

**Ejercicio 7.8 — Cambiar el jugador que comienza.** Para permitir que partidas consecutivas alternen el jugador que comienza, modifique la función miembro *vaciar* del tipo *Tablero*. Esta función reinicia el tablero sin fichas y establece el turno al jugador contrario al que empezó. *Nota: si conoce el número de fichas insertadas y el turno actual, podrá deducir el jugador que comenzó, aunque si desea despreocuparse de este cálculo siempre puede insertar un nuevo atributo para recordar este dato.*

## Jugador automático

Una modificación interesante es la inclusión de un jugador automático, es decir, que en lugar de solicitar la columna donde insertar se escoja de forma automática. Lo que conocemos como “jugar contra la máquina”.

Una solución ambiciosa podría incluir establecer distintos niveles de inteligencia, lo que podría resolverse cambiando incluso la interfaz del módulo; por ejemplo, creando un jugador automático de una forma especial que incluya el nivel de inteligencia a desarrollar o introduciendo funciones miembro que cambien o establezcan los niveles del jugador.

No vamos a entrar en algoritmos de inteligencia artificial, sino en el problema de modificar nuestro programa para que incluya esta capacidad. Nos bastará con un jugador automático que seleccione una columna de forma aleatoria con un mínimo de cambios. La propuesta es codificar la automatización del jugador en el nombre. Si el nombre comienza con el carácter '@' es que el jugador es automático.

**Ejercicio 7.9 — Incluir un jugador automático.** Modifique la función *escogeColumna* para que incluya la posibilidad de un jugador automático. Para ello, comprobará si el nombre comienza por '@', en cuyo caso seleccionará cualquier columna de forma aleatoria. *Nota: tenga en cuenta que si la columna ya está completa debería seleccionar otra.*

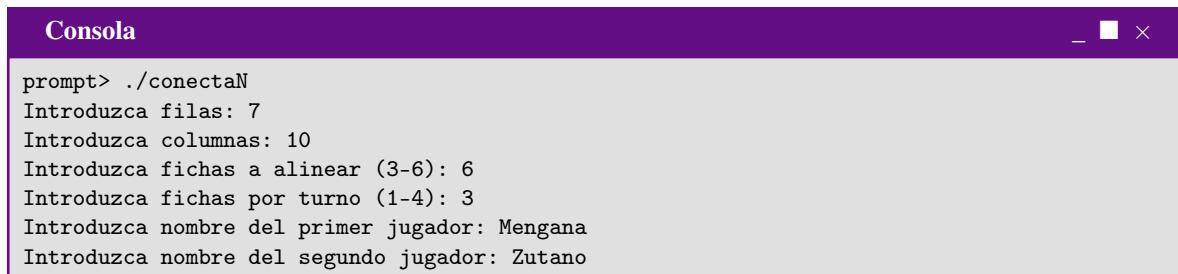
## Cambiar estrategia de turnos

Una vez que se ha desarrollado el juego y puesto en práctica, se ha determinado que en las partidas con un número de fichas a alinear alto es difícil conseguir ganar. Para aliviar este problema, se ha decidido cambiar las reglas. Concretamente, se ha decidido que en cada turno el jugador puede insertar más de una ficha. Es decir, el tablero está definido por unas dimensiones, un número de fichas a alinear y un número de fichas a insertar por turno.

Para implementar esta estrategia tendremos que realizar varios cambios en el programa:

- Cuando creamos un tablero con la configuración deseada, tendremos también que añadir un parámetro que indique el número de fichas por turno. Este parámetro no puede ser superior a  $N - 2$ . Por ejemplo, en un *conecta-5*, el valor estará en el rango [1, 3].
- Será necesario incluir alguna o algunas funciones de consulta para el tipo *Tablero*. Por ejemplo, para saber el número de fichas que corresponden a cada turno o las fichas que restan al turno actual.
- Cuando comienza el juego, también hay que preguntar el número de fichas por turno. Tenga en cuenta que no todos los valores son válidos.
- Se deberá modificar la E/S del tipo *Tablero* para que incluya la nueva información asociada.

Por ejemplo, un comienzo de partida con este tipo de configuración puede ser el siguiente:



```

prompt> ./conectaN
Introduzca filas: 7
Introduzca columnas: 10
Introduzca fichas a alinear (3-6): 6
Introduzca fichas por turno (1-4): 3
Introduzca nombre del primer jugador: Mengana
Introduzca nombre del segundo jugador: Zutano
  
```

Con esta configuración, cada jugador tendrá que insertar 3 fichas para pasar el turno. Por ejemplo, podemos seleccionar la primera columna:

Consola

```
a b c d e f g h i j
| | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |
=====
Turno de jugador 1: (x)
Mengana, escoja una columna (letra a-j) para la ficha 1 de 3: a
```

Lo que implica que el programa vuelve a solicitar la siguiente columna al mismo jugador:

Consola

```
a b c d e f g h i j
| | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |
| | x | | | | | | |
=====
Turno de jugador 1: (x)
Mengana, escoja una columna (letra a-j) para la ficha 2 de 3:
```

Y sólo cuando se han insertado tres fichas se solicita la inserción al segundo jugador:

Consola

```
a b c d e f g h i j
| | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |
| | | x | | | | | | |
| | | x | | | | | | |
| | | x | | | | | | |
=====
Turno de jugador 2: (o)
Zutano, escoja una columna (letra a-j) para la ficha 1 de 3:
```

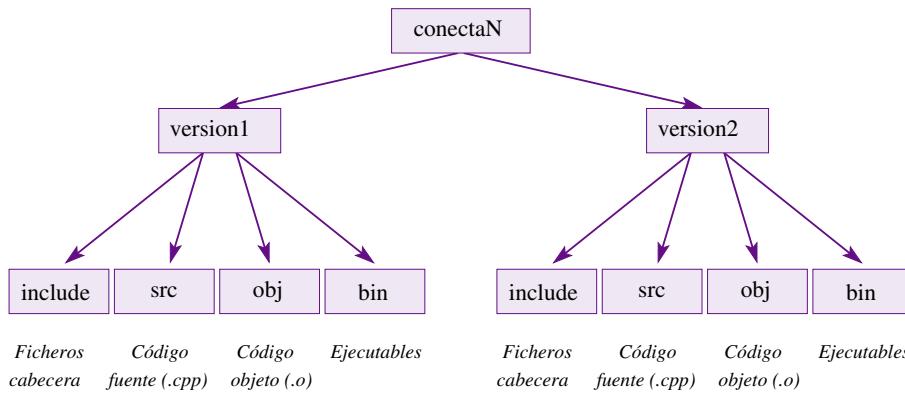
**Ejercicio 7.10 — Modificar estrategia de turnos.** Modifique el programa para que cada jugador pueda insertar un número de fichas variable.

## 7.5 Práctica a entregar

Debe crear una carpeta —por ejemplo, `conectaN`— en la que incluya todos los archivos que componen la solución de este guión. Tenga en cuenta que debe crear dos versiones, por lo que deberá tener dos directorios —por ejemplo, `version1` y `version2`— que contendrán dos programas completamente independientes. En la figura 7.5 se presenta un esquema de cómo quedarán. Observe que no hay un directorio para bibliotecas, pues se generará el ejecutable directamente a partir de los archivos objeto.

Aunque en este guión se han descrito las tareas de modificación como una secuencia de ejercicios, la segunda versión no contendrá ninguna referencia a ellos, sino la solución final. Tenga en cuenta que la secuenciación de tareas sólo se ha planteado para guiar al estudiante hasta la solución final.

Para desarrollar la solución, puede crear la primera versión y una vez terminada, copiarla exactamente en otro directorio. El archivo `Makefile` y toda la estructura que tenía será válido para realizar las modificaciones para la segunda versión. Además, no olvide que deberá incluir operaciones que permitan la “*limpieza*” de archivos intermedios generados.



**Figura 7.5**  
Directorios del proyecto *conectaN*.

Para poder empaquetar el resultado de este proyecto, es recomendable que realice una “*limpieza*” para eliminar los archivos temporales o que se pueden generar a partir de los fuentes. Una vez eliminados, sítuese en la carpeta superior y use la orden **tar** para obtener el archivo resultado. Más concretamente, ejecute lo siguiente:

**Consola**

```

prompt> cd version1
prompt> make clean
prompt> cd ../version2
prompt> make clean
prompt> cd ../..
prompt> tar zcvf conectaN.tgz conectaN
    
```

tras lo cual, dispondrá de un nuevo archivo **conectaN.tgz** que contiene la carpeta **conectaN**, así como todos los archivos y directorios que cuelgan de ella.



```

#include <iostream>
#include "fecha.h"
using namespace std;

Fecha CentroPeriodo(const Fecha& f1, const Fecha& f2)
{
    return f1+(f2-f1)/2;
}

void Intercambiar(Fecha& f1, Fecha& f2)
{
    Fecha aux;
    aux=f1; f1=f2; f2=aux;
}

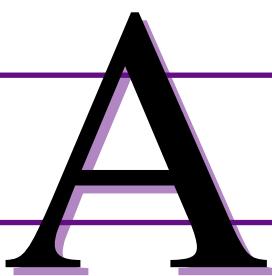
int main()
{
    Fecha f1,f2;

    cout << "Introduzca la fecha inicial: ";
    cin >> f1;
    cout << "Introduzca la fecha final: ";
    cin >> f2;

    if (f1>f2) {
        cerr << "¡Cuidado: la primera fecha no es anterior!" << endl;
        Intercambiar(f1,f2);
    }

    cout << "El periodo tiene " << f2-f1 << " días." << endl;
    cout << "El periodo se sitúa aproximadamente el "
        << CentroPeriodo(f1,f2) << endl;
}

```



# Guía de Estilo

## Introducción.....87

## Indicaciones generales.....87

Énfasis automático

Reglas de estilo

## Guía de estilo.....88

Identificadores

Estructura del código

Consideraciones adicionales

```

if (is >> f >> c)
if (f << 0 || c << 0) {
    is.setstate(std::ios::failbit);
return is;

```

## A.1 Introducción

La legibilidad de un programa es una característica fundamental al evaluar la calidad de un código escrito en cualquier lenguaje. Es importante no sólo si queremos que otros programadores entiendan mejor nuestros programas, sino para que nosotros mismos podamos manejarlos más fácilmente, por ejemplo, en tareas de mantenimiento donde tengamos que revisar código que hemos escrito hace meses o años.

Cuando los programas crecen e incluyen un mayor número de características, la variedad de tipos de elementos que pueden aparecer hace que una guía de estilo sea, no sólo recomendable, sino verdaderamente necesaria. Es importante seguir una guía de estilo que sea respetada por los distintos programadores que van a trabajar con el programa. Además, teniendo en cuenta que en muchos casos no podemos prever quién va a revisarlo, deberíamos adaptarnos a criterios generales ya establecidos en el pasado y ampliamente usados.

A pesar de esta necesidad, nadie ha llegado a establecer ningún estilo concreto para el lenguaje C++. Podemos decir que ningún estilo es perfecto y, por tanto, el programador debería ajustarse al estilo que se haya asumido para el proyecto en el que está trabajando. Podemos encontrar estilos propuestos por autores de prestigio —como *B. Stroustrup* o *Kernighan and Ritchie*— o estilos establecidos por distintas empresas, pudiendo destacar las grandes empresas como *Google*, *Apple* o *Microsoft*.

Es esta guía se listan algunos consejos que pueden ser útiles si tiene poca experiencia y que le pueden guiar hacia un buen estilo que mejore la legibilidad de sus programas. Además, le permitirá acercarse a un estilo propio de otros programadores experimentados, facilitando compartir código y trabajar en proyectos comunes donde las distintas aportaciones deberían respetar un estilo común.

Tenga en cuenta que nuestro objetivo no es tanto crear “código bonito”, sino código que facilite la programación y el mantenimiento del programa. Note que la belleza aquí no es un concepto estético sino más bien algo mucho más práctico, como es de esperar en una ingeniería. Podríamos incluso usar herramientas automáticas de ofuscación de código para crear bonitas figuras<sup>1</sup> que corresponde a código prácticamente imposible de entender.

Por consiguiente, las reglas sobre escritura de código no se alinean tanto con la estética, sino con una razón práctica que hace que el código sea más eficaz como implementación de un algoritmo que debe ser leída, entendida y modificada a lo largo del tiempo. Use esta guía no para seguirla ciegamente, sino para reflexionar sobre los motivos que hacen que un código que sigue cierto estilo se convierte en un código mejor.

## A.2 Indicaciones generales

El objetivo fundamental en una guía de estilo es maximizar la legibilidad del código que se escribe. Esta optimización facilita las tareas de creación, reparación y modificación posterior, especialmente cuando se realizan por distintas personas o en grupos de trabajo. Nuestro interés será establecer una forma de escribir que permite identificar rápida y fácilmente tanto los distintos tipos de elementos que forman el programa como las estructuras lógicas que los relacionan.

<sup>1</sup>Puede consultar “código ofuscado” en la red para descubrir múltiples programas y resultados.

## A.2.1 Énfasis automático

La primera recomendación para escribir y leer código es usar un editor especializado. Este editor puede realizar tareas de énfasis automático del código. Por ejemplo, en su versión más simplificada podríamos usar un editor que enfatiza las palabras que corresponden a palabras claves del lenguaje. La mayoría de los editores de texto incluyen sistemas automáticos para que el texto se adapte al lenguaje o contexto al que se refiere el documento. Si su editor muestra un aspecto plano y no es capaz de realizar esta tarea, debería buscar una alternativa (existen múltiples editores de código abierto que pueden descargarse para esto).

Un editor más elaborado podría fácilmente ayudar en otras tareas, como enfatizar literales: números enteros, números en coma flotante, caracteres o cadenas de caracteres. Además, podría identificar algunos componentes simples como directivas de precompilación, comentarios o bloques de sentencias.

La versión más avanzada y la más recomendable es usar un entorno de desarrollo especializado para el lenguaje: un **IDE** (*integrated development environment*). Normalmente, estos entornos incluyen editores de código que no sólo tienen en cuenta características simples como las indicadas, sino que son capaces de analizar estructuras más complejas del lenguaje. Algunos aspectos que el programador agradecerá en este tipo de software son por ejemplo:

- *Identificación de errores de programación.* Puede no sólo mostrar la sintaxis sino indicar cuándo esta sintaxis es incorrecta. Por ejemplo, puede subrayar estructuras escritas que no tendrán sentido para el compilador, o incluso analizar el código y determinar algunos avisos como que un identificador se usa antes de tener un valor asignado.
- *Podrían ayudarnos a adaptar nuestro código a una guía de estilo.* Cuando escribimos el código, el editor nos resalta las distintas partes de las estructuras que escribimos o inserta espacios automáticamente. Incluso puede incluir alguna herramienta para seleccionar un trozo de código y reescribirlo según alguna guía sintáctica simple.
- *Ayuda en la escritura del programa.* Puede incluir consejos o mostrar alternativas para ayudar a que el programador pueda escribir el código con más rapidez y menos probabilidad de fallo. Por ejemplo, al llamar a una función puede sugerirnos los parámetros que necesitaremos o cuando seleccionamos un componente de una estructura listarnos los campos entre los que, inevitablemente, el lenguaje nos obliga a escoger.

Estos aspectos no son propiamente parte de una guía de estilo. Considere esta breve exposición como una invitación a seleccionar un editor —o mejor un entorno de desarrollo— adecuado para trabajar.

## A.2.2 Reglas de estilo

Las reglas de estilo que vamos a proponer se refieren principalmente a dos aspectos:

1. *Enfatizar la estructura de nuestro programa*, dejando claro cuáles son los módulos que lo componen y cómo se pueden separar en distintos componentes, desde los bloques más generales —bibliotecas o ficheros— hasta los componentes más básicos; pasando por los tipos de datos, sus componentes, funciones, sentencias de selección, bucles, bloques de instrucciones e incluso cada parte de una instrucción. En particular, será de especial interés mostrar claramente la relación secuencial o de anidamiento que refleje el flujo del programa.
2. *Enfatizar el papel de cada uno de los componentes del programa*. Este aspecto se refiere fundamentalmente a cómo nombrar cada uno de esos elementos. Estos nombres deberían identificar claramente tanto su naturaleza —un objeto simple, una variable global, una función, un tipo de dato, etc.— como el papel que tiene en el programa.

En las siguientes secciones listamos algunas alternativas y consejos para crear un estilo propio para este curso. Por tanto, será el estilo que usemos en el código de nuestros ejemplos y el que se aconseja al estudiante. Cuanto antes se acostumbre, antes se beneficiará de sus ventajas.

No entienda que un programador debe usar siempre un determinado estilo, pues la elección dependerá también de otros factores. Además, siempre podremos adaptarnos a nuevos estilos, dependiendo del tipo de proyecto que desarrollemos. Sin embargo, incluso como programador individual, será conveniente establecer un estilo adecuado (por ejemplo, con el simple uso de determinado editor “inteligente” o *IDE*, podríamos modificar algunos aspectos de estilo). A pesar de todo, las guías generales de distintos estilos no son tan distintas, pues todas pretenden los mismos objetivos. Seguro que será fácil que se adapte a nuevos estilos en éste y en otros lenguajes.

## A.3 Guía de estilo

En esta sección vamos a listar los distintos consejos que componen la guía de estilo que proponemos en este curso. Se establece una guía que justifica los listados del curso y permite compartir código con el mismo estilo, lo que hace más fácil el trabajo en grupo y el desarrollo del curso, sin olvidarnos de que el objetivo final es motivar al estudiante y prepararlo para poder adaptarse a cualquier guía de estilo en un grupo o proyecto.

Como consecuencia, la exposición no será una larga lista de reglas que deben respetarse para conseguir un código homogéneo, sino una exposición razonada de algunas alternativas que motiven al estudiante a establecer un buen estilo. Lo dividimos en tres bloques: los dos primeros sobre identificadores y estructura del código, y un tercero donde añadiremos algunos consejos útiles con los últimos detalles.

### A.3.1 Identificadores

Al desarrollar un programa es necesario nombrar los distintos objetos que lo componen, desde el nombre de los archivos, incluyendo nuevos tipos de datos, funciones o métodos, objetos y variables de tipos básicos. Seleccionar cuidadosamente un identificador nos facilitará entender el papel que juega en el programa. Para ello, como contexto general debemos tener en cuenta aspectos como:

- Longitud variable de los identificadores.
- Podemos usar letras, dígitos y el carácter '\_'.
- Distinguir entre mayúsculas y minúsculas.
- Prefijos/posfijos.

#### Longitud de los identificadores

Recordemos que la longitud de un identificador puede ser muy variable. Nuestro compilador nos permite seleccionar nombres que van desde un carácter a varias decenas. ¿Cuál es la más adecuada?

Una característica fundamental de un nombre es que debe mostrar claramente qué contiene. En principio, especialmente para programadores con poca experiencia, esto es una invitación a crear identificadores largos que, si bien pueden ser muy ilustrativos para programas simples, no son de uso generalizado. Para aclarar esta situación, podemos sopesar distintos aspectos como:

- Con un identificador corto hay que escribir menos.
- Los identificadores cortos son más propensos a coincidir.
- Los identificadores largos pueden incluir más información sobre su semántica.
- Los identificadores largos pueden oscurecer el código, especialmente por el tamaño final de las sentencias. Por ejemplo, considere una expresión compleja en la que aparezcan varios de ellos.
- Debemos distinguir claramente entre distintos identificadores. Dos identificadores muy cortos o muy largos que se parezcan mucho hacen más difícil la lectura.

Teniendo en cuenta estas reflexiones, podemos proponer las siguiente regla de estilo:

**Regla A.1 — Longitud de los identificadores.** La longitud de los identificadores debería ser relativamente corta, intentando simplificar su longitud manteniendo tanto su significado como la distancia con respecto a otros identificadores.

Para aclarar las consecuencias de esta regla, veamos algunos ejemplos más concretos:

- La longitud de un identificador puede de alguna forma verse afectada por el tamaño del ámbito donde se conoce. En ámbitos muy locales podemos usar identificadores muy cortos, mientras que en ámbitos muy grandes deberían ser de mayor longitud, con un significado más preciso y con menos posibilidades de confusión.  
Un ejemplo son las variables contador que se usan en los bucles, donde es habitual usar los nombres *i*, *j*, *k* para indicar un entero que controla el bucle. Recuerde que el ámbito probablemente se limite al cuerpo del bucle.
- No deberíamos abusar de los acrónimos a no ser que sean ampliamente usados y bien conocidos. Un acrónimo reduce el tamaño, pero lo hace más fácil de confundir y más difícil de interpretar. Sin embargo, algunos son de amplio uso y son una buena opción. Por ejemplo, podemos usar el tipo *ColorRGB* para guardar un determinados color usando una tripleta de valores *RGB* (*Red*, *Green*, *Blue*).
- Debemos evitar tamaños grandes, especialmente si dan lugar a identificadores muy parecidos. Por ejemplo, el siguiente código es poco legible y propenso a errores:

```
double factor ponderacion_dato1; // Demasiado largos
double factor ponderacion_dato2;
double resultado_final ponderado;
//...
resultado_final ponderado= factor ponderacion dato1*dato1 + factor ponderacion dato2*dato2;
```

especialmente teniendo en cuenta que podríamos escribir un trozo equivalente mucho más simple y legible como:

```
double peso1; // Cortos y precisos
double peso2;
double resultado;
//...
resultado= peso1*dato1 + peso2*dato2;
```

- Deberíamos evitar añadir palabras y longitud que no contribuyan especialmente a aclarar el papel del identificador. Por ejemplo, podemos prescindir de los artículos como en:

```
double interes del prestamo; // Demasiada literatura
int el numero de componentes;
```

siendo igualmente claro y más corto usar nombres como:

```
double interes_prestamo;
int numero_componentes;
```

- Podemos incluir abreviaturas y nombres cortos siempre que no den lugar a ninguna ambigüedad. Por ejemplo, podemos usar nombres como:

```
double x_izq, x_drcha; // Abreviado pero preciso
double y_inf, y_sup;
```

incluyendo algún nombre simple y genérico si realmente el código no necesita más aclaración:

```
void Intercambiar(int& a, int& b)
{
    int aux(a); // Mínimo, pero más que suficiente
    a= b;
    b= aux;
}
```

## Letras y dígitos

Los identificadores pueden incluir también dígitos. Esta opción nos amplía la gama de posibilidades que tenemos pero, si tenemos en cuenta las consideraciones que hemos presentado en las secciones anteriores, deberían usarse sólo si consigue crear nombres diferenciados y mejorar el significado.

**Regla A.2 — Dígitos en identificadores.** No abuse del uso de dígitos en los identificadores, especialmente si con ello puede dar lugar a nombres parecidos y no contribuyen especialmente al significado del nombre.

Por ejemplo, podemos proponer un código en el que haya una amplia variedad de nombres como en:

```
double a1, a2, a3, a4; // Poco significativo
double b0, b0, b1, b1; // Difícil de leer
```

Podemos considerar una excepción a esta regla si realmente el problema que resolvemos enumera claramente distintos objetos. Por ejemplo, si queremos resolver una ecuación de segundo grado, las soluciones serán con alta probabilidad como las siguientes:

```
double x1, x2;
```

o si preferimos algo más de significado al nombre, por ejemplo porque se van a usar en un ámbito más amplio y queremos precisar mejor su función:

```
double raiz1, raiz2;
```

Por otro lado, recordemos que no es posible usar la letra “ñ”. Para este caso, el mejor consejo es evitar los nombres con esta letra o usar alguna forma con sonido similar. Por ejemplo, podemos usar:

```
int dia, mes, anio;
```

**Regla A.3 — Letra ñ.** La letra “ñ” no es válida, por lo que es recomendable evitar nombres que la incluyan, ya que no es posible usar el correspondiente identificador. En caso de que sean especialmente útiles, podemos aproximar el nombre con un sonido similar.

## Minúsculas y mayúsculas

La elección no se limita únicamente a escoger entre todo minúscula y todo mayúscula. También podemos intercalar minúsculas y mayúsculas así como el carácter ‘\_’ como separador. En general, podemos plantear distintos estilos que pueden usarse para enfatizar las propiedades que caracterizan al identificador correspondiente.

**Regla A.4 — Estilos con minúsculas y mayúsculas.** Aproveche la variedad de estilos con el uso de mayúsculas y minúsculas para caracterizar el tipo o naturaleza de los identificadores.

Algunos estilos ampliamente utilizados en programación son los siguientes:

- Identificadores en minúscula. Podemos escribir todas las letras en minúscula. En el caso de varias palabras podemos separarlas con el carácter ‘\_’. Por ejemplo:

```
finalizado, umbral_bajo, indice_maximo, caracter_final
```

- Identificadores en mayúscula. Podemos escribir todas las letras en mayúscula, también con la posibilidad de usar el carácter ‘\_’ como separador. Por ejemplo:

```
MAXIMO, CUADRADO, UMBRAL_MAXIMO
```

- Primera letra en mayúscula (puede encontrar este estilo con el nombre *Pascal case*). Todas las palabras que componen el identificador comienzan con mayúscula. En este caso no es necesario un separador. Por ejemplo:

```
Color, Hipotenusa, ResolverPorBiseccion, MatrizCuadrada
```

- Primera letra del identificador en minúscula y la primera de cada palabra concatenada en mayúscula (puede encontrar este estilo con el nombre *Camel case*). De nuevo, no es necesario un separador. Por ejemplo:

```
hacerNulo, maximoValor
```

En nuestro caso, proponemos un estilo sencillo e intuitivo en el que la globalidad o importancia de un identificador se resalta con el uso de mayúsculas, mientras que los objetos más locales se escriben en minúscula. Veamos algunos ejemplos y propuestas de estilo concretas.

En primer lugar, es importante destacar el caso de identificadores escritos enteramente con mayúsculas. En español estamos acostumbrados a usarlas para destacar alguna palabra o en determinados casos, siendo lo más habitual usar las mayúsculas sólo en la primera letra. Realmente, la lectura de texto completamente en mayúsculas resulta incómoda y poco recomendable.

En lenguaje C es común usar los identificadores completamente en mayúsculas para las macros (definidas con `#define`). En lenguaje C++ sigue siendo una regla generalizada que nosotros asumimos.

**Regla A.5 — Macros.** Los identificadores de las definiciones o macros con la directiva `#define` se escribirán enteramente en mayúscula.

Un ejemplo de estas definiciones o macros puede ser el siguiente:

```
#define MAXIMO 100
#define CUADRADO(x) ((x)*(x))
```

Un caso particular de objetos que se definen con esta directiva en C son las constantes globales. En el caso de querer fijar un valor concreto en tiempo de compilación, en C se puede resolver con `#define` de forma que el precompilador sitúa ese valor en cada posición donde se introduce el identificador. Es una solución muy eficiente<sup>2</sup> que asocia una constante global con la directiva `#define` y que por tanto se escribe en mayúsculas.

**Regla A.6 — Constantes fijadas en tiempo de compilación.** Los identificadores de las constantes fijadas en tiempo de compilación, ya sean definidas con `#define` o mediante la palabra reservada `const`, se escribirán enteramente en mayúscula.

Por ejemplo, las siguientes constantes tienen un valor fijado en tiempo de compilación y que sabemos que no cambiarán durante la ejecución del programa. Por tanto, se escribirán en mayúscula:

```
const double UMBRAL_BAJO= 0.0;
const double UMBRAL_ALTO= 1.0;
```

Sólo las macros y constantes usarán el estilo de todo mayúscula. El resto de identificadores usarán algunas de las otras formas de estilo que hemos listado, donde las mayúsculas se asociarán de alguna forma a identificadores más globales o de mayor importancia.

**Regla A.7 — Variables de ámbito local.** Los identificadores de objetos que no sean globales se escribirán con minúsculas, separando las palabras con el carácter `'_'`.

El caso de los tipos de datos o las funciones globales pueden tener un ámbito más amplio y normalmente se usarán en lugares distantes del código, lo que sugiere que el nombre debe ser escogido con cuidado para que sea fácilmente distinguible y destaque frente a otros nombres.

**Regla A.8 — Tipos y funciones globales.** Los identificadores de tipos definidos por el usuario y de funciones globales usarán las mayúsculas, incluyendo la primera de las letras (*Pascal Case*).

Finalmente, podemos distinguir un caso adicional: los métodos o funciones miembro. Si bien son también funciones, en realidad están subordinadas a una clase, lo que nos invita a poder distinguirlas de las funciones globales.

**Regla A.9 — Métodos o funciones miembro.** Los identificadores de métodos o funciones miembro se escribirán en minúscula. En el caso de que se componga de varias palabras, las siguientes se escribirán en mayúscula (*Camel Case*).

## Prefijos y posfijos

Los prefijos y posfijos predeterminados pueden permitirnos codificar información adicional sobre las propiedades o naturaleza de un identificador, siendo el caso de los prefijos de especial interés al ser más visibles en la lectura.

Uno de los acuerdos o convenciones más conocidos a la hora de crear un prefijo es el uso de la llamada *notación húngara*<sup>3</sup>. En gran medida, la difusión de esta notación se debe especialmente a que se creó dentro de *Microsoft* y se ha usado ampliamente por sus programadores. Sin embargo, no debe considerarse una garantía, pues actualmente muchos programadores la desaconsejan<sup>4</sup>.

En general, la notación húngara hace referencia a codificar el tipo o propósito como prefijo del nombre de una variable. Más concretamente, se puede hablar de dos tipos de notaciones:

1. Notación *húngara de sistemas*. Se codifica el tipo de la variable, por lo que la lectura del nombre conlleva conocer directamente este tipo. La ventaja es que tenemos más detalles en el contexto donde se usa, facilitando por ejemplo detectar incompatibilidades o errores.

<sup>2</sup>Aunque también se pueden considerar otros inconvenientes o desventajas frente al uso de un objeto definido con `const`.

<sup>3</sup>El origen del nombre se debe al creador de la notación, *Charles Simonyi*, con relación a su origen —nació en Hungría— y su lengua natal.

<sup>4</sup>Incluso *Microsoft* en algunas de sus especificaciones llega a indicar explícitamente que no se debe usar.

2. Notación *húngara de aplicaciones*. Se codifica el propósito o la lógica de la variable. Realmente no nos interesa el tipo al que pertenece, aunque es posible que se pueda deducir a partir de dicha lógica.

En la práctica, muchos programadores aconsejan no usar esta notación, aunque realmente la mayoría de estas recomendaciones son especialmente para el primer caso<sup>5</sup>. En realidad, la propuesta de *Simonyi* estaba más bien en la línea de la notación *húngara de aplicaciones*, que tiene más sentido en lenguajes de alto nivel, como C++.

Dada la controversia sobre su uso, no vamos a asumir el uso normativo de esta notación, aunque la presentamos para mostrarla como una opción que podría permitirnos idear nombres que puedan ser más significativos; siempre en la línea de notación de aplicaciones, rechazando por completo la codificación del tipo de dato en el nombre de la variable, que se ha mostrado claramente poco aconsejable.

**Regla A.10 — Notación húngara.** Se pueden añadir prefijos para codificar o enfatizar cierta información sobre alguna propiedad o propósito de una variable. Se desaconseja la codificación del tipo de dato concreto en el nombre.

Como ejemplos habituales que puede encontrar y que probablemente estén en la línea de este tipo de notación, podemos distinguir:

- Usar alguna letra simple para indicar el propósito de la variable. Por ejemplo, se puede usar la letra '*p*' inicial para indicar que es un puntero, la letra '*n*' para indicar un número de elementos, o las letras '*x*' e '*y*' para indicar que son coordenadas en los ejes correspondientes.
- Usar algún prefijo para indicar que una variable es un miembro de una clase. Por ejemplo, es habitual encontrar códigos donde se añade el carácter '*\_*' como prefijo, o incluso el par de caracteres '*m\_*' (*m* de miembro).
- Un prefijo para destacar alguna cualidad de un tipo definido. Por ejemplo, una clase diseñada como interfaz podría contener una primera '*I*' antepuesta al nombre.

Respecto al uso del carácter '*\_*' es interesante destacar que normalmente se desaconseja su uso como primer carácter de un identificador. El motivo es que incluso desde los primeros años de C se ha desarrollado mucho código para el que se asume cierta implicación entre este prefijo y su uso privado. Si se asume esta relación, es lógico aconsejar a los programadores que eviten comenzar los nombres con este carácter, garantizando que nunca coincidirán con nombres reservados. A pesar de ello, el hecho de que el lenguaje C++ garantice la encapsulación de los nombres dentro de una clase, abre la posibilidad de usar este tipo de prefijo sin que haya problemas de coincidencia.

Por ejemplo, podemos optar por añadir un prefijo a los atributos de una clase cuando tengamos nombres de atributos que coinciden con el nombre de una función miembro:

```
class Fecha {
    int _dia;
    // ...
public:
    int dia() const;
    // ...
};
```

Finalmente, es interesante añadir un último comentario sobre los *posfixos*. En la práctica, pueden pasar más fácilmente desapercibidos, por lo que no es habitual su uso. Sin embargo, es interesante destacar la forma de los identificadores que protegen de la inclusión múltiple de archivos cabecera. En este caso, se añade un posfijo que incluye la letra '*H*' —del inglés **Header**— para enfatizar su propósito de proteger un archivo de cabecera. Además, recuerde que como definición de precompilador se escribe enteramente en mayúscula y probablemente con un primer carácter '*\_*' que enfatiza su carácter interno o reservado.

### A.3.2 Estructura del código

Una vez seleccionados los *tokens* que componen nuestro programa, existen múltiples formas de escribirlos —unas mejores que otras— dependiendo de la forma en que los separemos y formateemos. En esta sección se presentan una serie de consejos que permitan obtener una estructura del código fácil de escribir, entender y modificar.

No existe una solución óptima al formateo del código. De ahí que existan múltiples guías de estilo sin que quede claro cuál es la mejor. A pesar de ello, comparten objetivos, pues todas persiguen obtener un buen listado que facilite el trabajo al programador. La forma de escribirlo debería representar de la mejor manera cuáles son los tokens, líneas, sentencias, bloques, módulos que componen el programa; debe representar correctamente la lógica del programa, informando de qué partes del código están relacionadas; debe facilitar una lectura rápida que permita detectar fácilmente errores. Todo esto teniendo en cuenta la limitación de usar un editor, donde podemos visualizar en un momento dado del orden de 100 líneas de 100 columnas; incluso menos si leemos en papel, por ejemplo, 50 líneas de 80 columnas.

#### Espacios en blanco

Las herramientas que tenemos para formatear el código son básicamente los espacios blancos, es decir, los caracteres como el espacio y el salto de línea. Tal vez considere también que contamos con el tabulador, pero antes incluso de empezar, lo descartamos. Ciertamente se puede usar, pero el hecho de que distintos editores puedan establecer distintos tamaños de tabulación invita a que se descarten y sustituyan por varios espacios.

<sup>5</sup>Aun así, algunos todavía pueden encontrar algunas ventajas especialmente cuando trabajamos a bajo nivel, en lenguaje C por ejemplo, donde tal vez sea especialmente importante conocer la codificación de tipos básicos.

Es posible que considere 8 espacios como un buen sustituto de tabulador, sin embargo cuando escribimos código C++ no es frecuente ese tamaño, pues acumular varios tabuladores llevaría el código demasiado a la derecha. Los valores típicos de sangrado no suelen pasar de 4. De hecho, si quiere evitar los tabuladores, lo mejor es configurar al editor para que haga el cambio directamente al pulsar el tabulador<sup>6</sup>. Si es un editor que permite la configuración para código, es probable que tenga dos opciones: una para cambiar el tabulador y otra para especificar exactamente cuántos son los espacios que determinan un nivel de sangrado.

Los espacios nos permiten separar elementos en la misma línea, con especial atención a los espacios iniciales, lo que denominamos *sangrado* de la línea<sup>7</sup>. En un programa se debe establecer un número fijo de espacios por cada nivel de sangrado.

**Regla A.11 — Sangrado.** El programa tendrá un número fijo de espacios por cada nivel de sangrado. Tamaños típicos son 2, 3 o 4.

Por otro lado, los saltos de línea nos permitirán separar las sentencias. Dos sentencias consecutivas deben escribirse en líneas separadas. Además, podemos usar dos saltos de línea para separar un bloque de sentencias relacionados o entidades de distinta naturaleza, como funciones o definiciones de tipos.

Por ejemplo, en el siguiente trozo de código hemos incluido 5 líneas que corresponden a un mismo nivel de sangrado, aunque se han separado por saltos de línea que muestran que el mensaje de lectura está relacionado con la operación sobre `cin`:

```
int main()
{
    double dato1= 0, dato2= 0;

    cout << "Introduzca dos números: ";
    cin >> dato1 >> dato2;

    double media= (dato1+dato2)/2;

    cout << "La media es: " << media << endl;
}
```

**Regla A.12 — Saltos de línea.** Dos sentencias distintas deben ir en líneas distintas. Se usarán dos saltos de línea para separar bloques o entidades diferenciadas, no más.

Observe que introducir demasiados saltos de línea nos puede llevar a un código que se extiende más que el tamaño de la pantalla, lo que dificulta su lectura.

### Sentencia simple

Una sentencia simple en una sola línea sería la guía fundamental que deberíamos seguir. Sin embargo, también tenemos que recordar que una única sentencia puede ser compleja, conteniendo múltiples elementos o incluso dando lugar a una línea demasiado larga.

Si la línea es muy larga, probablemente el editor nos permite escribirla haciendo que se salga por la parte de la derecha de la pantalla o forzando la visualización en varias líneas que se adaptan al ancho de la ventana.

**Regla A.13 — Líneas largas.** Se debe evitar dejar que el editor sea el responsable de formatear las líneas largas. Debemos ser nosotros los que formateemos la sentencia en varias líneas, enfatizando que es la misma mediante el sangrado, y dividiéndola por la parte que más facilite su lectura.

Note que es la mejor forma de garantizar la correcta visualización en cualquier editor y de establecer una partición que muestre la lógica que contiene. Incluso si tenemos una cadena de caracteres muy larga podemos modificar el código para que se corte en el lugar adecuado. Por ejemplo, es válido escribir:

```
cout << "Esto es un ejemplo de una cadena muy larga "
      "que se escribe en líneas separadas para que "
      "la lectura sea más sencilla e independiente "
      "del editor."
      << endl;
```

donde hemos especificado una única cadena de caracteres en distintas líneas.

Por otro lado, una sentencia puede llegar a ser muy compleja. Podemos usar los espacios para separar las distintas partes para que sea más fácil de leer.

<sup>6</sup>Tenga en cuenta que en C++ separar el código con un tabulador o un espacio no es relevante a la hora de compilar, por lo que podemos descartar directamente el uso de tabuladores. En otros lenguajes o reglas sintácticas tenga cuidado si el tabulador tiene un significado concreto. En este caso, el editor no debería sustituir tabuladores por espacios.

<sup>7</sup>Es posible que en algún sitio encuentre las palabras “indentar” e “indentación”, pero no dejan de ser anglicismos de las correspondientes en inglés: “indent” e “indentation”.

**Regla A.14 — Espacios en una misma línea.** Los espacios que separan los elementos de una línea deben enfatizar las distintas partes que la componen. En especial, se usarán para identificar subexpresiones.

Por ejemplo, la siguiente línea contiene una asignación en la que los espacios han separado claramente la variable a la que se asigna y las dos subexpresiones que forman la media ponderada:

```
resultado= peso1*dato1 + peso2*dato2;
```

En algunas guías de estilo puede encontrar una regla para separar todos los operandos y operadores de una expresión. El objetivo es separar más claramente los componentes, aunque en ese caso no aprovecha la posibilidad de separar subexpresiones, lo que podría facilitar la lectura en expresiones más complejas. Por eso hemos formulado una regla más laxa, donde cabe la separación completa en expresiones simples o el énfasis en la separación de subexpresiones.

### Sentencia compuesta

Una sentencia compuesta es un bloque de código dentro de un par de caracteres {}. Para enfatizar esta unión como sentencia compuesta, se escribirá en un nuevo nivel de sangrado.

**Regla A.15 — Sentencia compuesta.** Las sentencias compuestas se escribirán en un nuevo nivel de sangrado, incluyendo un salto de línea después de cada uno de los caracteres de apertura { y cierre }.

Lógicamente, el anidamiento de distintos bloques de código provocará la acumulación de niveles de sangrado.

### Estructuras de control

Podemos hablar del espaciado en las tres estructuras de control:

1. La estructura *secuencial*. Las sentencias se ejecutan una a una de manera independiente. Cuando se termina la ejecución de una sentencia, se pasa en la siguiente en la secuencia. Son sentencias independientes por lo que el espaciado refleja esta relación escribiendo las líneas consecutivas con idéntico sangrado.
2. La estructura *condicional*. En este caso tenemos una sentencia o bloque de sentencias subordinado, ya sea en la parte de condición verdadera o en la parte **else**. Esta relación se refleja aumentando el nivel de sangrado.
3. La estructura *iterativa*. El cuerpo del bucle se escribe aumentando el nivel de sangrado.

**Regla A.16 — Estructuras de control.** Como normal general, en la estructura condicional e iterativa, las sentencias que se incluyen dentro se escribirán con un nivel de sangrado superior. En el caso de ser un bloque definido entre llaves, la llave de apertura se escribirá al final de la primera línea, mientras que la última línea tendrá exclusivamente el cierre de llaves. El anidamiento de estructuras se muestra con la acumulación de niveles de sangrado.

Una opción también habitual es hacer que los bloques de sentencias usen una línea exclusivamente para la llave de apertura. Esta opción separa más las líneas y permite ver fácilmente qué llaves de apertura corresponden a qué llaves de cierre. Incluso en algún caso alguna guía también aconseja esta situación de llaves cuando hay una única sentencia, es decir, cuando las llaves son prescindibles.

Nosotros usaremos la versión más corta en número de líneas, intentando que en la pantalla podamos ver simultáneamente más código y donde no será difícil identificar el cierre de llaves con la instrucción correspondiente. En las secciones que siguen se especificará con ejemplos esta regla general, incluyendo alguna variación.

### Sentencia if-else

Tanto las instrucciones que están dentro del **if** como del **else** deberán estar sangradas en un nivel. El esquema que corresponde a la instrucción **if-else** es el siguiente:

```
if (condición)
    sentencia_if
else
    sentencia_else
```

que lógicamente podría aparecer sin la parte **else**. En el caso de usar bloques de sentencias, la sintaxis incluirá las llaves:

```
if (condición) {
    sentencias_if
}
else {
    sentencias_else
}
```

Observe que la palabra **else** está después del cierre de llaves. Algunas guías proponen aprovechar la línea anterior si contiene un cierre de llaves, aunque nosotros no lo haremos para visualizar más fácilmente el emparejamiento entre la parte **if** y la parte **else** y para separar más claramente los dos bloques de sentencias.

Por otro lado, es necesario especificar cómo se anidan las estructuras condicionales. Según la regla general que hemos establecido, basta con sangrar las instrucciones un nivel más. Por ejemplo, podemos escribir:

```
if (condición1) {
    // condición 1
```

```

    }
    else if (condición 2) {
        // condición 1 y condición 2
    }
    else {
        // condición 1 y no condición 2
    }
}

```

Note que el segundo **if–else** queda sangrado 5 espacios, para alinearlo a la derecha del primer **else**. Esta forma de escritura enfatiza especialmente cómo todo ese bloque está subordinado a que la primera condición sea falsa.

Otro ejemplo, algo más concreto donde se encadenan 3 instrucciones condicionales es el siguiente:

```

if (nota<5) {
    // Suspenso
}
else if (nota<7) {
    // Aprobado
}
else if (nota<9) {
    // Notable
}
else { // nota≥9
    // Sobresaliente
}

```

Observe que si el código es algo complejo, con varias estructuras anidadas podemos llegar a desplazar el código muy a la derecha. Una alternativa a este estilo es el que sigue:

```

if (condición1) {
    // condición 1
}
else if (condición 2) {
    // condición 1 y condición 2
}
else {
    // condición 1 y no condición 2
}

```

En general, usaremos el primer estilo, donde el anidamiento de espacios refleja más claramente la estructura del flujo de control. Sin embargo, también admitiremos este segundo, donde todos los bloques de sentencias se escriben sangrados un único nivel. Este segundo puede ser mejor alternativa cuando las condiciones de las sentencias **if–else** sean relativamente independientes, es decir, no sea especialmente importante enfatizar las dependencias entre ellas. Por ejemplo, en el siguiente trozo de código:

```

if (porcentaje≥0 && porcentaje≤10) {
    // Muy poco
}
else if (porcentaje≥40 && porcentaje≤60) {
    // Medio
}
else if (porcentaje≥90 && porcentaje≤100) {
    // Alto
}
else {
    // Sin intervalo claramente definido
}

```

las opciones podrían ser intercambiables, es decir, podríamos ordenarlas de otra forma sin que afectara a la lógica del programa. Note que esta forma de escritura refleja claramente cómo el flujo de control seleccionará uno de los 4 posibles casos. De alguna forma, recuerda a la selección múltiple.

**Regla A.17 — Sentencia if–else anidada.** En general se escribirá con un sangrado acumulativo que refleja cómo una condiciones se subordinan a otras. Opcionalmente, podremos usar el anidamiento en casos **else** al mismo nivel si queremos enfatizar un flujo de control que selecciona entre distintas posibilidades.

### Selección múltiple con **switch**

La selección múltiple en C/C++ siempre ha generado cierta controversia debido a que por un lado cada caso es un punto de salto y por otro aparece una instrucción **break** que rompe el flujo para salir del **switch**. La posibilidad de generar muchas situaciones, añadiendo o eliminando instrucciones **break** o el anidamiento con otros bloques que harían el código más confuso nos sugiere que es importante entender que la instrucción **switch** de alguna forma salta al caso seleccionado, y que la instrucción **break** rompe el flujo saltando al final.

Esta situación puede incomodar a los más puristas de la programación estructurada, e incluso sugerir que es preferible usar estructuras condicionales anidadas en su lugar, como las que hemos mostrado al final de la sección anterior. Sin embargo, es importante entender la capacidad que tiene esta instrucción para hacer mucho más legible una situación donde existen una serie de alternativas al valor de una expresión. En este caso, escribiremos la instrucción **switch** con el siguiente estilo:

```

switch (expresión) {
    case valor1:
        // opción valor1
        break;
    case valor2:

```

```
// opción valor2
break;
case valor3:
case valor4:
    // opción valor3 o valor4
    break;
default:
    // Si no es ninguna opción anterior
}
```

Note que hemos escrito explícitamente las instrucciones **break**, ya que normalmente vamos a implementar un flujo de control estándar donde no hay sorpresas, es decir, sin que haya casos de saltos inesperados o sentencias **break** en lugares poco habituales. De esta forma, a pesar de la aparente poco estructurado de la sentencia **switch**, nuestro código responde a un esquema habitual de selección múltiple en programación estructurada.

**Regla A.18 — Sentencia switch.** Se sangrará todo el bloque un nivel para luego sangrar cada uno de los casos. En cada uno de ellos, aparecerá una única instrucción **break** para cerrar el bloque.

### Bucles for y while

En estos bucles seguiremos la regla general, sangrando un nivel el cuerpo del bucle. En caso de un bloque de sentencias, la llave de apertura se escribe en la misma línea que la cabecera del bucle. Por ejemplo, a continuación presentamos dos bucles anidados:

```
for (inicialización; condición_for; incremento) {
    // ...
    while (condición_while) {
        // cuerpo de while
    }
    // ...
}
```

Un caso especial es el bucle vacío, donde la única instrucción del bucle es una sentencia vacía. En este caso, deberá escribirse en una nueva línea, como corresponde a la regla general:

```
for (inicialización; condición; incremento)
; // vacío
```

añadiendo opcionalmente un comentario que enfatice esta situación. Sin embargo, generalmente los bucles vacíos suelen ser poco legibles. En el caso del bucle **for** será preferible cambiar el bucle por uno no vacío. Este bucle puede escribirse como:

```
inicialización;
while (condición)
    incremento;
```

**Regla A.19 — Bucles vacíos.** Se evitarán los bucles vacíos. Un bucle **for** vacío se escribirá como el correspondiente **while** no vacío.

### Bucle do-while

El caso del bucle **do-while** será una excepción a la regla general. La llave de cierre no se escribirá sola sino que, en la misma línea, le seguirá la condición. El esquema será el siguiente:

```
do {
    // Cuerpo del bucle
} while (condición);
```

Además, siempre incluiremos la doble llave aunque sea una única sentencia. La razón de este tipo de estilo está en hacer más visible que es un bucle *post-test*, a diferencia del *pre-test* que también se especifica con la misma palabra **while**. Por ejemplo, observe el siguiente esquema:

```
// ...
while (expresión1) // ...
//...
//...
} while (expresión2);
//...
```

En este trozo de código, un solo vistazo a cualquiera de las líneas con la palabra **while** ya nos deja claro qué tipo de bucle es el que se está usando. En el primero sabemos que el cuerpo del bucle está debajo y en el segundo sabemos que buscando hacia arriba encontraremos el **do** que abre el bucle.

**Regla A.20 — Bucle do-while.** El bucle **do-while** siempre contendrá un bloque entre llaves, aunque sea una única sentencia. Además, la condición se añadirá en la misma línea que el cierre de llaves.

## Funciones globales

En el caso de las funciones globales, la llave de apertura del bloque del cuerpo se escribe en una línea independiente:

```
double Hipotenusa(double c1, double c2)
{
    return sqrt(c1*c1+c2*c2);
}
```

En este caso ocupamos más líneas, aunque si tenemos en cuenta que el número de funciones puede ser relativamente pequeño y que la implementación de una función es un bloque de código independiente del código que le precede y le sigue, este estilo es una buena opción. De hecho, la mayoría de las propuestas de estilo incluyen esta forma de escribir las funciones.

**Regla A.21 — Funciones globales.** La funciones globales se escribirán separando claramente la cabecera del cuerpo de la función. Para ello, el bloque de sentencias que la implementa se escribirá sangrado y reservando una línea tanto para la apertura como el cierre de llaves. Además, añadimos una línea vacía tanto antes como después de la función.

Note que si incluimos un comentario que especifica el funcionamiento de la función junto a la cabecera, la interfaz quedará claramente separada de la implementación.

## Espaciado en una estructura/clase

En C++ existe una fuerte relación entre la palabra **struct** y la palabra **class**. En primer lugar, planteamos el estilo para el código que incluye una estructura al estilo C, es decir, con todos sus miembros públicos y sin funciones miembro. En este caso, el estilo es el siguiente:

```
struct Celda {
    int dato;
    Celda* sig;
};
```

donde vemos que sus campos se han sangrado para destacar que están dentro de la estructura. Note que los atributos se han listado en líneas separadas, lo que facilita añadir un comentario a la derecha si fuera necesario.

**Regla A.22 — Estructuras C.** Se usará el sangrado para listar los objetos miembro. La apertura de llaves se sitúa al final de la línea que tiene la palabra **struct** y el cierre de llaves en una nueva línea. Cada atributo se escribirá en una línea separada.

Aunque el lenguaje lo admite, no añadiremos la declaración de objetos antes del cierre —antes del punto y coma— de la definición. En lugar de hacer:

```
struct Celda {
    int dato;
    Celda *sig;
} * inicial;
```

optaremos por separarlos en:

```
struct Celda {
    int dato;
    Celda *sig;
};

Celda* inicial;
```

El estilo que usaremos para las clases —o las estructuras que se traten como clases, incluyendo secciones protegidas o privadas— será el siguiente:

```
class Matriz {
public:
    Matriz();
    // ...
private:
    int filas;
    int columnas;
    double* datos;
};
```

donde podemos ver que hemos sangrado las palabras clave **public** y **private** y a su vez cada uno de sus contenidos, ya sean atributos o métodos.

**Regla A.23 — Clases.** Se usa doble sangrado, uno para las palabras clave de control de acceso —**public**, **private** y **protected**— y un segundo sangrado para listar los objetos y funciones miembro. La apertura de llaves se sitúa al final de la línea que tiene la palabra **class** y el cierre de llaves en una nueva línea.

## A.3.3 Consideraciones adicionales

En esta sección incluimos algunos comentarios que si bien no se incluyen claramente dentro de las anteriores, son consejos útiles para crear un programa más legible y fácil de manejar.

## Tipo enumerado

El tipo enumerado resulta un caso especial ya que podemos distinguir entre el tipo enumerado de C o de C++98 y el nuevo tipo enumerado de C++11.

El primer caso se considera menos recomendable. Consiste en un tipo de dato que resuelve fácilmente el compilador asignando un valor de tipo integral a cada uno de los posibles valores. En la práctica, el nombre del tipo es equivalente a algún tipo integral —por ejemplo, el tipo `int`— y cada uno de los posibles valores es una constante. Este tipo de enumerado se puede usar tanto en C como en C++, incluso en el estándar C++11/14.

El estilo que usaremos para este tipo de enumerado será especificarlo completamente en una única línea. El nombre del tipo será con iniciales en mayúsculas —*Pascal Case*— mientras que cada uno de los valores se escribirá con todas las letras mayúsculas. Por ejemplo:

```
enum TonoGris { BLACK = 0, GRAY = 128, WHITE = 255};
```

donde hemos incluido, además, los valores concretos que se asignen a las tres constantes.

El hecho de escribir cada valor con todas mayúsculas se hace en coherencia a los criterios que hemos indicado antes. Si tenemos en cuenta que se comportan como constantes y que el tipo se declara de forma global, podemos verlas como constantes enteras globales<sup>8</sup>.

Los principales inconvenientes de este tipo enumerado es que los valores se comportan como constantes por lo que podrían realizarse conversiones automáticas entre el tipo *TonoGris* y tipos enteros. Además, las constantes podrían coincidir con otros identificadores.

Para evitar estos inconvenientes, en C++11 se proponen los tipos *enumerados fuertemente tipados* o tipos *enumerados con ámbito*. Un ejemplo es el siguiente:

```
enum class ColorEstado { Verde, Naranja, Rojo };
```

donde hemos añadido la palabra `class` para indicar que es un tipo enumerado fuertemente tipado. En este caso, el compilador también aplica una solución similar asignando un tipo integral para sustentar el valor almacenado<sup>9</sup>, pero evita las conversiones automáticas y que los valores sean identificadores en el ámbito global. Un código válido que usa este tipo es:

```
ColorEstado luz = ColorEstado::Verde;
```

donde puede ver que el valor *Verde* se debe especificar dentro del ámbito de *ColorEstado*.

**Regla A.24 — Valores de enumerados.** Los valores de los tipos enumerados serán en mayúsculas cuando queramos reflejar su papel como variable global. En caso de querer enfatizar el papel de alternativa de un tipo enumerado, independiente de valores constantes, escribiremos cada valor con las iniciales en mayúsculas (*Pascal Case*). En este caso, se aconseja el uso de enumerados fuertemente tipados.

Por tanto, la forma en que se escribirán los valores de un tipo enumerado será variable, ya sea para enfatizar que estamos interesados a usar el valor constante correspondiente o que estamos interesados en representar un tipo enumerado propiamente dicho, donde el valor asignado a cada alternativa no es relevante.

Finalmente, es interesante indicar que podríamos especificar los valores en distintas líneas, aunque lo habitual es usar una sola. En caso de que haya muchos valores, o que los valores se puedan clasificar en distintos subgrupos o que queramos documentar el sentido de cada valor, se separarán en distintas líneas.

## Palabra reservada `class` o `struct`

En C++ es posible usar tanto la palabra `class` como la palabra `struct` para definir un nuevo tipo de dato que incluye objetos y funciones miembro. Básicamente, la única diferencia es el control de acceso por defecto, `private` y `public` respectivamente.

**Regla A.25 — Struct vs class.** Se usará la palabra `struct` cuando el nuevo tipo de dato contenga únicamente atributos y métodos públicos. En caso de contener un miembro con acceso restringido, se usará la palabra reservada `class`.

Para mejorar la legibilidad del código, usaremos la palabra reservada `struct` para enfatizar que no queremos encapsular nada. Es decir, que todos los miembros, ya sean atributos o métodos, serán públicos. En el momento en que queramos ocultar algo, por ejemplo un atributo como `private`, optaremos por la palabra reservada `class`.

## Ordenar `private`, `protected`, `public`

El lenguaje no requiere un orden para las distintas secciones del control de acceso en una clase. De hecho, incluso podrían repetirse y mezclarse sin que afectara al resultado. Esta situación implica una decisión sobre cómo situarlas.

Para casos complejos podría haber distintas alternativas, pero para clases sencillas como las que aparecen en este curso no hay muchas posibilidades. En principio, podemos plantear dos órdenes: `private-protected-public` o al revés.

<sup>8</sup>De hecho, este tipo de enumerados se han usado y se usan en C++ como un “truco” muy simple para crear una constante entera en un ámbito determinado.

<sup>9</sup>No siempre lo selecciona, pues tenemos la posibilidad de especificar el tipo concreto que hay debajo e incluso el entero que se asigna a cada valor.

**Regla A.26 — Ordenar `public`, `protected`, `private`.** El orden de las secciones de distinto control de acceso para este curso se hará comenzando con la parte privada.

En principio, el comportamiento por defecto de la palabra `class` invita a empezar con la parte privada. Además, dado que en el curso centramos las discusiones en cómo implementar una clase es lógico comenzar el diseño pensando en la representación que irá en la parte privada. Desde este punto de vista, nuestro estilo priorizará la parte privada.

Sin embargo, es importante indicar que no son razones determinantes. Algunos programadores consideran mucho más correcto priorizar la parte pública, ya que es la parte relevante para un programador que use la clase. Este usuario querrá encabezarla con la parte que realmente usará desde otros módulos.

## Declaraciones

Un aspecto que en principio parece poco relevante se refiere a cómo y dónde realizar las declaraciones de variables. Hay quien incluso lo simplifica diciendo que todas las variables deberían declararse al principio de la función donde se usen. Sin embargo, el lugar de declaración cambia si tenemos en cuenta que:

- No siempre se puede declarar una variable, pues tal vez no está disponible algún dato necesario para su inicialización.
- Las variables deberían declararse en el entorno más pequeño donde se usen, de forma que pensar en esa variable y el código que la usa se restrinja al mínimo número de líneas posible.

Tener todas las variables acumuladas en una zona hace más difícil localizar alguna concreta y facilita que pase desapercibido algún error oculto entre tanto tipo y nombre. No es mala idea tener un lugar donde encontrar las declaraciones, pero sí oscurecerlo con un exceso de datos e información de cosas que tal vez luego sean relativamente independientes.

En general, hay que evitar declarar varios objetos en una misma línea, a no ser que tenga una estrecha relación y esa declaración conjunta la enfatice. Por ejemplo, si queremos declarar dos coordenadas, podemos escribir:

```
double x, y;
```

en la misma línea, porque en el código que sigue la pareja de identificadores compondrá un punto concreto. Si dos nombres no tienen relación, una declaración separada los independiza y facilita añadir un comentario independiente para cada uno de ellos. Por ejemplo, una mala costumbre es declarar clasificando los identificadores por tipos —por ejemplo, enteros con enteros y reales con reales— intentando insertar un orden que no tiene que ver con la lógica del programa.

**Regla A.27 — Declaración.** Las variables se declararán en líneas independientes si no tienen una estrecha relación, situando dicha declaración en la zona más cercana posible al lugar donde se usa.

La separación en distintas líneas no sólo se refiere a distintas variables, sino que un solo objeto podría declararse en varias líneas para facilitar la lectura. No nos limitamos a ejemplos muy claros —como un vector o matriz inicializados que se escriben en distintas líneas— sino incluso cuando una declaración es poco legible. Por ejemplo, en lugar de declarar `operaciones` como sigue:

```
double (*operaciones[10])(double x, double y);
```

podemos aclararlo usando un sinónimo con `typedef`:

```
typedef double (*Pfunc)(double x, double y);
Pfunc operaciones[10];
```

## Punteros y referencias

Es interesante indicar que cuando definimos punteros o referencias también podemos definir un estilo para situar los caracteres `'*'` y `'&'`. Por ejemplo, las siguientes líneas usan distintos estilos:

```
int *puntero;
int* puntero;
int &referencia;
int& referencia;
```

Realmente cada una de las 4 podrían escribirse en un programa y significarían exactamente lo mismo. La duda surge especialmente porque la primera línea es la habitual en C. No es mala idea, porque nos recuerda que sintácticamente el asterisco hace que la variable sea de tipo puntero. Por ejemplo, en el siguiente código:

```
int *puntero, entero;
```

`puntero` es un puntero —de tipo `int*`— mientras que `entero` es de tipo `int` (sin puntero). Si queremos que la segunda sea un puntero, debemos repetir el asterisco. Por este motivo, gran parte del código que encontrará tiene este estilo.

Sin embargo, cuando leemos código en C++ con el tipo referencia, vemos que la mayoría de los programadores están más satisfechos con el otro estilo, a pesar de que podríamos plantear exactamente la misma justificación para poder declarar dos referencias en la misma línea.

Probablemente, la diferencia está en que es poco habitual encontrar una línea en la que se declaran dos referencias. Stroustrup sugiere que en C++ se haga más énfasis en el tipo, aconsejando que el asterisco esté junto al tipo y evitando declarar varios punteros en la misma línea.

**Regla A.28 — Punteros y referencias.** Los caracteres `'*'` y `'&'` se escribirán junto al tipo, evitando declarar varios punteros o referencias en la misma línea.

## Comentarios

Podemos escribir dos tipos de comentarios:

1. Los que ocupan líneas por sí solos.
2. Los que se escriben a la derecha del código.

En el primer caso, se usarán para documentar el código que viene a continuación, mientras que en el segundo documentan el código que queda a la izquierda.

Recuerde que el mismo código debería ser claro y legible, por lo que si respeta las reglas de estilo y la forma en que se nombran los componentes muchos comentarios serían redundantes. Es preferible minimizar la información, ya sea evitando comentarios innecesarios o evitando largos mensajes.

**Regla A.29 — Comentarios.** Los comentarios independientes —que ocupan líneas sin código— contienen información que documenta el código que viene a continuación y se escribirán respetando el sangrado. Los comentario a la derecha del código contienen información sobre esa línea. En ambos casos, los mensajes deberán ser concisos y no redundantes.

## Compilación separada

El programa se escribe en distintos archivos de texto con una extensión que indica que contiene código C++, ya sea un archivo de cabecera o un archivo de implementación. Tenemos distintas alternativas que puede consultar en el manual de su compilador. Por ejemplo, puede encontrar extensiones como `.h`, `.hpp`, `.hh`, `.cpp`, `.cxx`, `.c++`, `.tcc`, etc.

**Regla A.30 — Nombre de archivos.** Los nombres de archivos se escribirán en minúscula, usando el alfabeto inglés y el carácter `'_'` en caso de ser compuestos. Las extensiones serán `.h` para los de cabecera y `.cpp` para los archivos a compilar. Cada archivo `.cpp` dará lugar a un archivo compilado que se enlazará para obtener el ejecutable. La directiva `#include` se usará exclusivamente para archivos de cabecera.

Note que esta regla define claramente cómo se llamarán los archivos y cómo se van a relacionar en el proceso de compilación. Realmente, la directiva `#include` puede incluir cualquier archivo, incluso los `.cpp`. Dado que esta guía pretende ayudar a un programador que comienza, es importante dejar claro cómo se diseña la compilación separada, aunque si avanza a otros conceptos —como las plantillas en C++— verá que esta regla debería completarse con nuevas situaciones.

Por otro lado, tal vez le parezca muy restrictivo el uso del alfabeto inglés y el carácter separador, pero si piensa que queremos obtener código que sea portable, se dará cuenta de que es la mejor forma de garantizar que no habrá ningún problema cuando los lleve a un sistema donde no sabemos cómo se codificarán otros caracteres o si un espacio intermedio entre palabras lo va a considerar separador y por tanto interpretará una pareja de nombres de ficheros.

Una vez decididos los nombres de los archivos, los de cabecera se insertarán con la directiva `#include`. En caso de que haya más de uno, podríamos incluirlos en cualquier orden, ya que el código que hayamos escrito en un archivo cabecera debería funcionar independientemente del resto del código que le preceda. Sin embargo, lo ideal es situar primero los archivos cabecera del sistema o de bibliotecas externas ya terminadas.

**Regla A.31 — Orden de archivos cabecera.** Los archivos de cabecera se situarán al comienzo del archivo que los incluye, situando en primer lugar los externos y a continuación los del proyecto que se están desarrollando priorizando los más básicos que estén ya cerrados.

El motivo está en la simplificación de los mensajes de error que se generan. Si incluimos primero uno de nuestros archivos con un error y luego un archivo del sistema o de una librería externa terminada podríamos obtener una larga lista de errores situados en los archivos externos. Recordemos que si el compilador encuentra un error en nuestro archivo cabecera seguirá analizando el resto del código para completar la lista de errores. Incluso podríamos obtener algún caso más complejo si nuestro código genera una situación incorrecta que se detecta en el código posterior: tendríamos nuestro código con un error y una lista de errores que comienza en código externo.

El único motivo que podría justificar priorizar nuestros archivos cabecera sería que queremos garantizar que el código es válido cuando se incluye en primer lugar. Por ejemplo, si tenemos un archivo cabecera que necesita incluir `iostream` pero no lo hace, podríamos compilar sin problemas si en los archivos `cpp` se incluye siempre antes que nuestro archivo.

**Regla A.32 — Archivo cabecera compilable.** Un archivo de cabecera debe ser compilable independientemente del orden de inclusión. En especial, debemos garantizar que es compilable si es el primer archivo incluido, es decir, la unidad de compilación comienza con ese archivo cabecera. Además, deberá admitir la inclusión múltiple.

Recuerde que la inclusión múltiple se controla mediante directivas del precompilador, comenzando el fichero con la directiva `#ifndef` y terminando con `#endif`. Note que si el archivo de cabecera tiene a su vez una directiva `#include`, es

recomendable que se sitúe también dentro de este entorno. Lógicamente, si están bien diseñados, también sería válido incluir el archivo antes de `#ifndef`, pero esta inclusión conllevaría más trabajo para el precompilador.

Finalmente, la sintaxis para incluir un archivo admite dos posibilidades, incluir el nombre con los caracteres `<>` o con los caracteres `" "`. Los primeros provocarán la búsqueda del archivo en los directorios predeterminados y los segundos añadirán a esos directorios el lugar donde se encuentra el archivo fuente. Note que si configuramos el entorno y las opciones del compilador adecuadamente, podríamos usar los caracteres `<>` en todos los casos sin que haya errores.

**Regla A.33 — Sintaxis de inclusión.** Los archivos externos se incluirán con los “ángulos dobles” `<>`, mientras que los archivos propios del proyecto se incluirán con las dobles comillas `" "`.

Usar las comillas simplifica la tarea de compilación, ya que si todos los archivos están en el mismo sitio, no es necesario añadir directorios donde buscar para que el compilador encuentre los archivos de cabecera. Probablemente piense que esta justificación no es especialmente relevante, ya que es fácil configurar el entorno o las llamadas al compilador para que busquen archivos cabecera en otros directorios del proyecto.

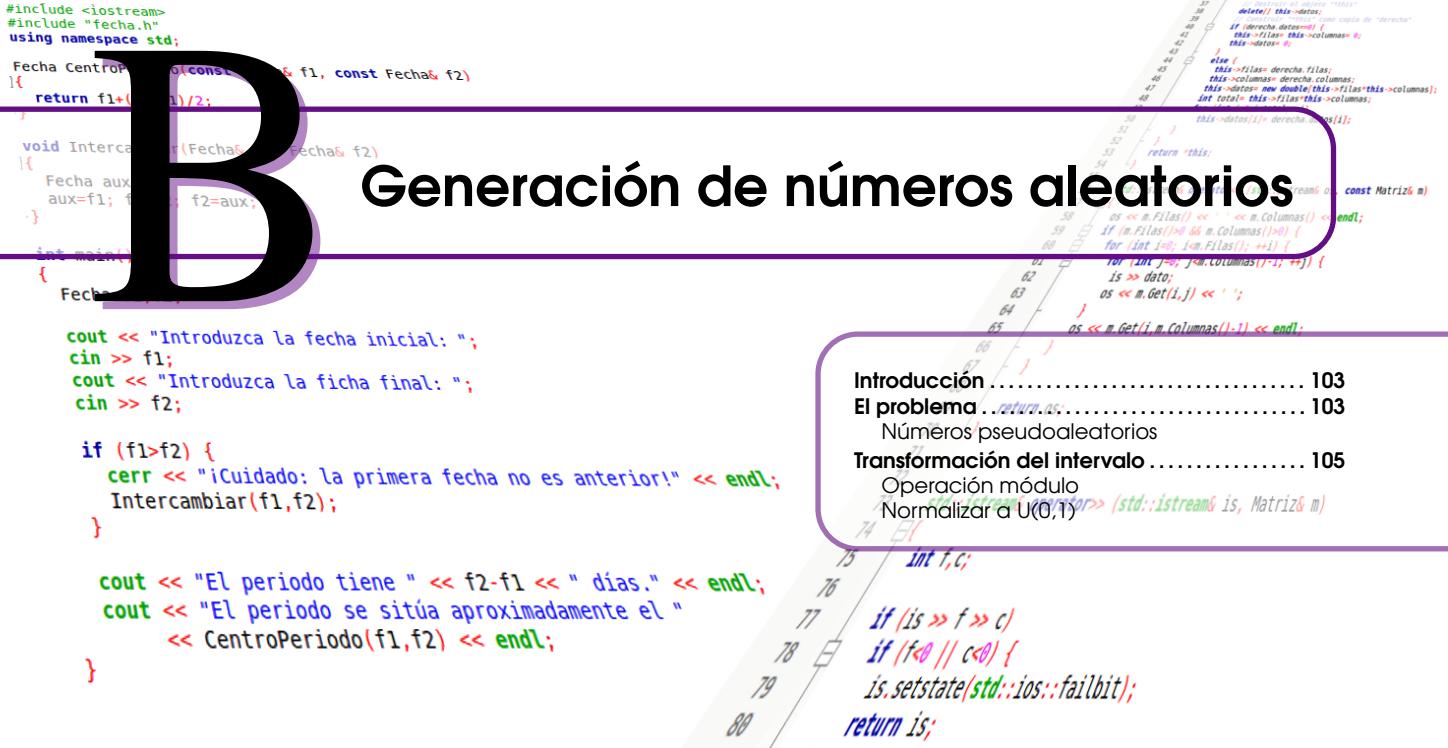
Efectivamente, es fácil resolverlo, aunque la razón fundamental por la que proponemos esta regla es la legibilidad: nos va a permitir ver más claramente qué archivos son externos y qué archivos son parte del proyecto.

### Otras consideraciones

Para terminar la guía, incluimos algunas consideraciones cortas que no vale la pena extender porque ya se conocen en parte o que no pueden incluirse fácilmente dentro de las reglas de estilo que hemos presentado en las secciones anteriores. Algunos consejos son:

- Evite las variables globales.
- Evite el código no estructurado, especialmente el uso de `goto`, `continue` y `break`, excepto dentro de la sentencia `switch` como se ha descrito en las secciones anteriores.
- No abuse del uso de `return` para salir directamente de una función. Sin embargo, no debería considerar su uso como una ruptura de las reglas de la programación estructurada y por tanto un uso incorrecto. Evalúe si su introducción simplifica el código resultante.
- Intente obtener funciones cortas. En concreto, intente evitar que haya bloques de código tan grandes que las llaves de apertura y de cierre disten más de una pantalla.
- No abuse de las macros, pues cuando lea su código no estará viendo realmente lo que se compila. De hecho, se desaconsejan para definir constantes o macros que pueden definirse con `const` y funciones `inline`, respectivamente.
- Intente eliminar los avisos —`warnings`— del compilador modificando el código.
- Evite un diseño que conlleve un encadenamiento de llamadas excesivamente largo. Recuerde que seguir la ejecución de un programa implica saltar en el código cada vez que hay una llamada.
- Aunque en este curso se ha usado intensivamente, no abuse de los recursos de bajo nivel que ofrece C++. El conjunto de posibilidades que ofrece el lenguaje es muy amplio, pero no tiene sentido usar herramientas de bajo nivel para implementar algoritmos de alto nivel. La solución debe adaptarse al tipo de aplicación que está programado. Si está escribiendo un programa en C++, probablemente puede hacerlo con la *STL*, sin necesidad de gestionar recursos con punteros, memoria dinámica, etc.





## B.1 Introducción

Muchos programas incluyen la generación automática de números aleatorios. Un ejemplo muy claro es un juego donde se tiene que avanzar con eventos que simulen aleatoriedad a fin de obtener cierta variedad en el desarrollo de distintas partidas.

Por ejemplo, imagine que deseamos crear un programa que avanza en un juego donde se lanza un dado. El programa simulará que obtenemos valores del conjunto  $\{1, 2, 3, 4, 5, 6\}$  hasta el final de la partida. Por ejemplo, podemos obtener:

1, 1, 6, 2, 4, 4, 3, 6, 2, 5, 5, 5, 1, ...

donde puede ver que se obtienen distintos valores que se pueden repetir con un orden indeterminado. Lógicamente, cuando empecemos una nueva partida, los valores que se obtienen deben ser distintos a los de la partida anterior.

En este apéndice vamos a ver cómo podemos hacer que nuestros programas puedan generar dichos valores. Para ello presentamos las funciones que están disponibles en C++ y que ya se podían usar en lenguaje C.

Existen otras utilidades disponibles sólo en lenguaje C++ —desde el estándar C++11— que resultan especialmente interesantes, no sólo porque puede realizar las mismas operaciones, sino porque ofrecen herramientas más avanzadas para crear fácilmente soluciones a problemas más complejos. Si bien no son difíciles de manejar, para entender bien estas nuevas herramientas convendría conocer algunos conceptos de teoría de la probabilidad. No es intención de este documento entrar en esa teoría, por lo que presentaremos las ideas más básicas de una forma muy intuitiva de forma que pueda resolver los problemas más simples y habituales sin gran dificultad.

## B.2 El problema

El problema consiste en que un programa tiene que ser capaz de generar una secuencia de números aleatorios tan larga como deseemos. Es decir, estamos interesados en obtener una serie de valores aleatorios:

$x_0, x_1, x_2, \dots, x_i, \dots$

Para disponer de una utilidad básica que nos resuelva este problema basta con crear una función que nos devuelva cada uno de esos valores conforme la llamamos. En lenguaje C, y por tanto en C++, se ofrece la solución más simple: crear una función **rand** que devuelve estos valores. Esta función no tiene parámetros. Sólo devuelve un nuevo valor entero aleatorio. Por ejemplo, si deseamos obtener 1000 valores aleatorios podemos escribir:

```
#include <cstdlib> // rand()
//...
for (int i=0; i<1000; ++i)
    cout << rand() << ' ';
```

La función está diseñada para obtener un valor entero en el rango  $[0, RAND\_MAX]$ , donde  $RAND\_MAX$  es una constante predeterminada. Tenga en cuenta que:

- Es necesario incluir el archivo **cstdlib** para disponer de esta función.
- La constante  $RAND\_MAX$  depende de su sistema, ya que no está fijada en el estándar. Sólo sabemos que es un entero positivo, ya que el valor devuelto por **rand** es **int**.

- El entero obtenido puede ser cualquiera del intervalo  $[0, RAND\_MAX]$  con igual probabilidad. Lo que se conoce por una distribución uniforme, ya que cualquier valor de ese intervalo, incluyendo el 0 y  $RAND\_MAX$ , se obtiene con igual probabilidad.
- Es de esperar que el valor de  $RAND\_MAX$  sea bastante alto. De hecho es posible que sea el entero más grande. Por ejemplo, en un sistema con tamaño de palabra pequeño, 2 bytes, podría tener el valor 32767 o si tiene un sistema de 32 bits, es probable que valga 2147483647.

### B.2.1 Números pseudoaleatorios

Como sabemos, el ordenador es una máquina determinista, es decir, que no tiene un comportamiento aleatorio, sino que las salidas son exactas y predecibles a partir de las entradas correspondientes. Por tanto, en principio es imposible conseguir generar una secuencia como la indicada anteriormente, es decir una secuencia de valores realmente aleatorios.

A pesar de ello, y gracias a los estudios estadísticos que se han realizado, existen métodos relativamente simples para obtener una secuencia que *parezca aleatoria*. Efectivamente, en la práctica, la mayoría de los problemas necesitan que los números parezcan aleatorios, es decir, que sean números que analizados estadísticamente podamos decir que se comportan como aleatorios. A éstos los llamaremos *pseudoaleatorios*.

No vamos a entrar en detalles de cómo funciona la función `rand`, pero para entender su uso podemos decir que los números se generan según cierta función interna  $f(x)$  de manera que:

$$x_{i+1} = f(x_i)$$

Aunque parezca sorprendente, una idea tan simple y aparentemente tan poco aleatoria nos permite obtener la secuencia deseada. El truco está, lógicamente, en que no necesitamos números aleatorios, sino que basta con que lo parezcan. Ahora bien, todos los números están determinados por la función  $f(x)$ , pero no sabemos cuánto vale el primer valor con el que comienza la secuencia. Toda la secuencia depende de él, de manera que si fijamos un valor concreto, siempre obtendremos la misma secuencia. Por ello, se denomina *semilla*.

Si nuestros programas usaran siempre la misma semilla, los números pseudoaleatorios que generaríamos serían siempre los mismos. Si queremos que distintas ejecuciones den lugar a distintas secuencias, es necesario cambiar de semilla en cada ejecución. Una forma muy simple de obtener distintas semillas es usar el valor del reloj del sistema. Note que si ejecutamos dos veces distintas un mismo programa, la semilla dependería del momento en que damos la orden de ejecución.

Para fijar una semilla, usamos la función `srand` de `cstdlib`, y para obtener un valor del reloj del sistema la función `time` del fichero de cabecera `ctime`. Un programa muy simple que genera tres valores aleatorios es el siguiente:

```
#include <cstdlib> // rand, srand
#include <ctime> // time
using namespace std;
int main()
{
    srand (time(0));
    int aleatorio1= rand();
    int aleatorio2= rand();
    int aleatorio3= rand();
}
```

Observe que la semilla sólo hay que fijarla al principio del programa, una única vez. A partir de ese momento, se puede usar `rand` tantas veces como se desee para obtener nuevos valores pseudoaleatorios. Por ejemplo, si ejecuta el siguiente programa:

```
for (int i=0;i<10;++i) {
    srand (time(0));
    cout << rand() << ' ';
}
cout << endl;
```

es posible que obtenga 10 valores iguales. O tal vez, si por casualidad el valor de `time` avanza durante el bucle, 2 grupos con los valores iguales. Por ejemplo, una ejecución en mi máquina ha dado lugar a:

```
1094219464 1094219464 1094219464 1094219464 1094219464
1094219464 1094219464 1094219464 1094219464 1094219464
```

El problema es que hemos situado el generador en un valor de semilla idéntico en cada iteración. Situamos el primer valor, generamos el valor aleatorio, y en la siguiente iteración volvemos a situar de nuevo el generador en el primer valor. Recuerde que si `time` devuelve el valor en segundos, las 10 iteraciones del bucle probablemente situarán la misma semilla. Por tanto, el valor que se genera con `rand` será el mismo.

Realmente, `time` es muy poco aleatoria. Sin embargo, nosotros queremos un valor entero distinto cada vez que lancemos el programa, por lo que resolveremos el problema generando una primera semilla al comienzo y limitándonos a usar la función `rand` en el resto del programa. Note que la semilla dependerá del segundo en el que ejecutemos el programa, por lo que

resultará en ejecuciones con secuencias aleatorias distintas. Además, el que la secuencia parezca aleatoria está garantizado por la forma en que se comporta la función interna  $f(x_i)$ , es decir, si queremos que los valores de nuestro programa parezcan aleatorios, deberán corresponder a una secuencia generada con dicha función, una vez establecida la semilla.

## B.3 Transformación del intervalo

Normalmente no nos interesará generar un número aleatorio en el intervalo  $[0, RAND\_MAX]$ , especialmente teniendo en cuenta que no conocemos el valor de esa constante. Por ejemplo, si queremos lanzar un dado queremos un valor entero que va del 1 al 6. Para resolver el problema debemos transformar el valor generado al intervalo deseado.

### B.3.1 Operación módulo

Si deseamos obtener un valor en un intervalo de enteros pequeño lo más tentador es realizar una operación de módulo con el operador %. Por ejemplo, para generar el valor de un dado podemos escribir:

```
int dado= rand() % 6 + 1;
```

En general, si queremos obtener un valor en el intervalo de enteros  $[MIN, MAX]$ , ambos inclusive, podríamos calcularlo como:

```
aleatorio= rand() % (MAX-MIN+1) + MIN;
```

Es probable que si consulta código en la red, encuentre muchos ejemplos con líneas de este tipo. Este código se utiliza frecuentemente por su simplicidad, aunque no resulta especialmente recomendable cuando las características de aleatoriedad del generador son importantes para la aplicación que se está desarrollando.

Efectivamente, con la operación módulo estamos haciendo que el valor que usamos en nuestra aplicación dependa especialmente de los bits menos significativos del número generado. Por ejemplo, si realizamos una operación de módulo 100, realmente estamos cogiendo los dos últimos dígitos decimales de los enteros generados. Aunque el resultado final dependerá de la función  $f(x_i)$  que se esté usando como motor de generación, es probable que el comportamiento de los bits menos significativos sea menos aleatorio de lo esperado.

### B.3.2 Normalizar a U(0,1)

La variable aleatoria uniforme en el intervalo  $[0, 1]$  —que denotamos  $U(0, 1)$ — es especialmente importante en la teoría de la probabilidad. De hecho, si consulta distintos algoritmos de generación de números aleatorios para distintas distribuciones de probabilidad, encontrará que incluyen la generación de uno o varios valores de esta variable.

De forma simplificada, digamos que generar un valor de una variable  $U(0, 1)$  es obtener cualquier valor de ese intervalo, teniendo en cuenta que todos los números de ese intervalo tienen igual probabilidad. Con la función `rand` no tenemos más que transformar el valor dividiendo por el máximo `RAND_MAX`. En concreto, podemos usar la siguiente función:

```
inline double Uniforme01()
{
    return rand() / (RAND_MAX+1.0);
}
```

En este código debería observar que sumamos el valor 1.0, que es de tipo `double`. Es interesante notar que la operación `RAND_MAX+1` es peligrosa. Esta expresión es entera, por lo que la división del valor de `rand` entre este número sería entera y casi seguro que daría el valor cero. Podría pensar que el siguiente código resuelve el problema:

```
inline double Uniforme01()
{
    return rand() / (double) (RAND_MAX+1); // Error
}
```

Sin embargo, no sólo es peligrosa por eso, sino que el valor de `RAND_MAX` podría ser el entero más grande, por lo que al sumar uno se obtiene un entero incorrecto. Posiblemente, el entero más pequeño –el más negativo– del rango del tipo `int`.

Por otro lado, el valor que hemos obtenido es un número real del intervalo  $[0, 1)$ , sin llegar a alcanzar el valor 1.0. Con este número podemos obtener cualquier valor en el intervalo deseado sin más que realizar una transformación sencilla. Por ejemplo:

```
int dado= Uniforme01() * 6 + 1;
```

Al usar esta expresión el valor aleatorio se encuentra en el intervalo  $[0, 6]$  al multiplicarlo por 6, y en el intervalo  $[1, 7)$  al sumar 1. Cuando asignamos a la variable `dado`, nos quedamos con el valor entero correspondiente. En general, podemos obtener un valor en un intervalo con la siguiente función:

```
inline int Uniforme(int minimo, int maximo)
{
    double u01= rand() / (RAND_MAX+1.0); // Uniforme01
    return minimo + (maximo-minimo+1) * u01;
}
```



# Tablas

Tabla ASCII .....	107
Operadores C++ .....	108
Palabras reservadas de C89, C99, C11, C++ y C++11 .....	109
Manipuladores y funciones miembro para E/S formateada .....	109
Banderas y máscaras	
Funciones miembro y manipuladores	

## C.1 Tabla ASCII

En la figura C.1 se presenta la tabla de codificación ISO-8859-15 del alfabeto latino. Es similar a la ISO-8859-1 aunque difiere en 8 posiciones (`0xA4`, `0xA6`, `0xA8`, `0xB4`, `0xB8`, `0xBC`, `0xBD` y `0xBE`). Esta codificación se distingue especialmente porque incluye el carácter correspondiente al euro.

Aunque es probable que la codificación ISO-8859-15 no sea la que esté usando en su sistema, la mayoría de los problemas que se resuelven en los cursos de programación asumen que es la codificación para los caracteres o secuencias de caracteres.

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1x	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2x	SP	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6x	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL
8x	PAD	HOP	BPH	NBH	IND	NEL	SSA	ESA	HTS	HTJ	VTS	PLD	PLU	RI	SS2	SS3
9x	DCS	PU1	PU2	STS	CCH	MW	SPA	EPA	SOS	SGC	SCI	CSI	ST	OSC	PM	APC
Ax	NBSP	í	¢	£	€	¥	Š	§	š	©	¤	«	¬	SHY	®	-
Bx	°	±	²	³	Ž	µ	¶	·	ž	¹	º	»	Œ	œ	Ý	ÿ
Cx	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
Dx	Đ	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	þ	ß
Ex	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
Fx	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

**Figura C.1**  
Tabla de codificación ISO-8859-15.

Observe que todos los caracteres especiales que existen en distintos idiomas y no en inglés están en la segunda parte de la tabla. Realmente, la tabla ASCII propiamente dicha es la primera mitad, mientras que la segunda es una extensión. Por ello, esta tabla no es la tabla ASCII, sino la tabla ASCII extendida ISO-8859-15.

En la práctica aceptaremos esta codificación para nuestras soluciones, ya que nos interesa especialmente el algoritmo a resolver sin entrar en detalles sobre los problemas de codificación. Note que podría tener problemas en la ejecución de

algoritmos en español incluso si su sistema usa esta codificación. Por ejemplo, si quiere comprobar el orden de dos letras para ordenar una cadena, los caracteres con tilde están fuera del rango del alfabeto '*a*'-'*z*'.

Si su sistema usa otras codificaciones como ISO-8859-1, Windows-1252, o UTF-8, no tendrá problemas en ejecutar y comprobar el funcionamiento de sus algoritmos si evita el uso de la parte extendida, ya que en esas codificaciones los caracteres básicos se representan exactamente igual. Podríamos decir que realizamos soluciones que funcionan sin ningún problema solamente en inglés.

Más adelante es posible que tenga que resolver problemas para distintos lenguajes. Tendrá que consultar el tipo de codificación de su sistema y qué posibilidades tiene para resolverlo. Las soluciones pueden ir desde un tipo **char** con otra codificación hasta usar bibliotecas de funciones que resuelven sus problemas configurando el lenguaje usado.

## C.2 Operadores C++

En la siguiente tabla se listan los operadores de C++. Los operadores situados en el mismo recuadro tienen la misma precedencia.

En general, para no tener ningún problema con la asociatividad, intente recordar que los operadores unarios y de asignación son asociativos por la derecha, mientras que los demás lo son por la izquierda.

Si revisa el estándar, podrá encontrar que realmente los unarios postfijo son de izquierda a derecha así como el operador condicional, aunque en la práctica no suelen crear incertidumbre.

Operadores de C++		
Operador	Nombre	Uso
::	Global	:: <i>nombre</i>
::	Resolución de ámbito	<i>espacio_nombres</i> :: <i>miembro</i>
::	Resolución de ámbito	<i>nombre_clase</i> :: <i>miembro</i>
->	selección de miembro	<i>puntero</i> -> <i>miembro</i>
.	Selección de miembro	<i>objeto</i> . <i>miembro</i>
[]	Índice de vector	<i>nombre_vector</i> [ <i>expr</i> ]
()	Llamada a función	<i>nombre_función</i> ( <i>lista_expr</i> )
()	Construcción de valor	<i>tipo</i> ( <i>lista_expr</i> )
++	Post-incremento	<i>valor_i</i> ++
--	Post-decremento	<i>valor_i</i> --
<b>typeid</b>	Identificador de tipo	<b>typeid</b> ( <i>type</i> )
<b>typeid</b>	... en tiempo de ejecución	<b>typeid</b> ( <i>expr</i> )
<b>dynamic_cast</b>	conversión en ejecución con verificación	<b>dynamic_cast</b> < <i>tipo</i> >( <i>expr</i> )
<b>static_cast</b>	conversión en compilación con verificación	<b>static_cast</b> < <i>tipo</i> >( <i>expr</i> )
<b>reinterpret_cast</b>	conversión sin verificación	<b>reinterpret_cast</b> < <i>tipo</i> >( <i>expr</i> )
<b>const_cast</b>	conversión const	<b>const_cast</b> < <i>tipo</i> >( <i>expr</i> )
<b>sizeof</b>	Tamaño del tipo	<b>sizeof</b> ( <i>tipo</i> )
<b>sizeof</b>	Tamaño del objeto	<b>sizeof</b> <i>expr</i>
++	Pre-incremento	++ <i>valor_i</i>
--	Pre-decremento	-- <i>valor_i</i>
~	Complemento	~ <i>expr</i>
!	No	! <i>expr</i>
+	Más unario	+ <i>expr</i>
-	Menos unario	- <i>expr</i>
&	Dirección de	& <i>valor_i</i>
*	Desreferencia	* <i>expr</i>
<b>new</b>	reserva	<b>new</b> <i>tipo</i>
<b>new</b>	reserva e iniciación	<b>new</b> <i>tipo</i> ( <i>lista_expr</i> )
<b>new</b>	reserva (emplazamiento)	<b>new</b> ( <i>lista_expr</i> ) <i>tipo</i>
<b>new</b>	reserva (con inicialización)	<b>new</b> ( <i>lista_expr</i> ) <i>tipo</i> ( <i>lista_expr</i> )
<b>delete</b>	destrucción (liberación)	<b>delete</b> <i>puntero</i>
<b>delete</b> []	... de un vector	<b>delete</b> [] <i>puntero</i>
()	Conversión de tipo	( <i>tipo</i> ) <i>expr</i>
.*	Selección de miembro	<i>objeto</i> .* <i>puntero_a_miembro</i>
->*	Selección de miembro	<i>puntero</i> ->* <i>puntero_a_miembro</i>
*	Multiplicación	<i>expr</i> * <i>expr</i>

continúa en la página siguiente

continúa de la página anterior		
Operador	Nombre	Uso
/	División	<i>expr/expr</i>
%	Módulo	<i>expr%expr</i>
+	Suma	<i>expr+expr</i>
-	Resta	<i>expr-expr</i>
<<	Desplazamiento a izquierda	<i>expr&lt;&lt;expr</i>
>>	Desplazamiento a derecha	<i>expr&gt;&gt;expr</i>
<	Menor	<i>expr&lt;expr</i>
<=	Menor o igual	<i>expr&lt;=expr</i>
>	Mayor	<i>expr&gt;expr</i>
>=	Mayor o igual	<i>expr&gt;=expr</i>
==	Igual	<i>expr==expr</i>
!=	No igual	<i>expr!=expr</i>
&	Y a nivel de bit	<i>expr&amp;expr</i>
~	O exclusivo a nivel de bit	<i>expr^expr</i>
	O a nivel de bit	<i>expr expr</i>
&&	Y lógico	<i>expr&amp;&amp;expr</i>
	O lógico	<i>expr  expr</i>
?:	expresión condicional	<i>expr?expr:expr</i>
=	Asignación	<i>valor_i=expr</i>
*=	Multiplicación y asignación	<i>valor_i*=expr</i>
/=	División y asignación	<i>valor_i/=expr</i>
%=	Resto y asignación	<i>valor_i%=expr</i>
+=	Suma y asignación	<i>valor_i+=expr</i>
-=	Resta y asignación	<i>valor_i-=expr</i>
<<=	Desplazar izq. y asignación	<i>valor_i&lt;&lt;=expr</i>
>>=	Desplazar der. y asignación	<i>valor_i&gt;&gt;=expr</i>
&=	Y y asignación	<i>valor_i&amp;=expr</i>
^=	O exclusivo y asignación	<i>valor_i^=expr</i>
=	O y asignación	<i>valor_i =expr</i>
throw	Lanzamiento de excepción	<i>throw expr</i>
,	Coma	<i>expr ,expr</i>

Tabla C.1  
Operadores de C++

Estos operadores están incluidos en el estándar C++98. En C++11 aparecen tres más:

`sizeof..., noexcept, alignof`

### C.3 Palabras reservadas de C89, C99, C11, C++ y C++11

Es interesante conocer las partes comunes del lenguaje C y C++. En esta sección presentamos la tabla C.2 con las palabras reservadas de las distintas versiones de ambos lenguajes. La parte más interesante se encuentra en los dos primeros bloques, donde se incluyen las palabras reservadas que estaban presentes en C cuando se creó C++. Dado que éste “heredó” gran parte de los contenidos de C, el segundo bloque de C++ se presenta como una adición a las del primero (comunes a ambos).

A pesar de ello, los últimos tres bloques presentan palabras que se han ido añadiendo en las sucesivas versiones de los lenguajes. Aunque todavía muchos programadores creen que C++ es un lenguaje ampliación de C, lo cierto es que los dos evolucionan de forma independiente.

### C.4 Manipuladores y funciones miembro para E/S formateada

En la siguiente figura C.2 se presentan los manipuladores y funciones miembro que ofrece C++ para formatear la E/S. Es recomendable estudiar la forma en que funcionan antes de usar esta tabla, ya que se presenta más como una tabla de referencia que como una lista de posibilidades.

Fundamentalmente, la E/S está controlada por un conjunto de banderas y valores almacenados en el flujo:

- Las banderas no son más que un valor booleano que indica si una característica está activa o no. Por ejemplo, hay una bandera para indicar si se presenta el signo a los números positivos. Por defecto sólo se presenta el signo a los negativos —con el carácter ‘-’— pero si activamos la bandera `showpos`, también se incluirá el carácter ‘+’ cuando el número sea positivo.
- Se incluyen valores para controlar otros aspectos más complejos. Por ejemplo, el flujo almacena el carácter con el que tiene que llenar el espacio que queda cuando ajustamos un campo a la derecha.

**Tabla C.2**  
Palabras reservadas de C89, C99, C11, C++ y C++11.

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
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		
Añadidas en C99			
_Bool	_Complex	_Imaginary	inline
restrict			
Añadidas en C++11			
alignas	alignof	char16_t	char32_t
constexpr	decltype	noexcept	nullptr
static_assert	thread_local		
Añadidas en C11			
_Alignas	_Alignof	_Atomic	_Generic
_Noreturn	_Static_assert	_Thread_local	

#### C.4.1 Banderas y máscaras

Si observa la figura C.2 (página 112) descubrirá que no sólo hablamos de banderas, sino también de *máscaras*. Las máscaras no son más que un grupo de banderas. Son necesarias para poder realizar fácilmente operaciones sobre todas las banderas que componen la máscara.

Por ejemplo, la base usada al escribir un entero se controla con tres banderas: *oct*, *dec*, *hex*. Con tres banderas tenemos hasta 8 posibilidades dependiendo de cuáles activamos o no. Sin embargo, realmente sólo tienen sentido cuatro: primera activa, segunda activa, tercera activa o ninguna activa. No tiene sentido indicar que la salida se formatea en octal y decimal, por ejemplo.

Para poder manejar fácilmente un grupo de banderas en el que sólo tiene sentido que una esté activa se crean las máscaras. Estas máscaras son la unión de las distintas banderas. Cuando activamos una bandera de una máscara, realmente estamos desactivando también las otras.

A modo ilustrativo, presentamos el siguiente esquema de código para que entienda este diseño y le resulte más sencillo entender el mecanismo que usamos. En primer lugar definimos un entero que representa el lugar donde vamos a guardar todas las banderas.

```
// Un entero para almacenar hasta 32 banderas
unsigned int estado; // Está en algún lugar definido.
```

Este entero podría estar definido en algún lugar interno al flujo. Con esta definición como un simple entero, sólo gastaremos 4 bytes para controlar todas las banderas, hasta 32.

Para que el usuario —nosotros— podamos usar las banderas, se crean algunas constantes con nombres de banderas y máscaras. Por ejemplo, podemos crear los 3 nombres siguiente:

```
const int MascaraTriple= 0x7; // Una máscara que agrupa los 3 bits situados al final del estado
const int Bandera1= 0x1;      // Una de las banderas incluidas en la máscara
const int Bandera2= 0x2;      // Una de las banderas incluidas en la máscara
const int Bandera3= 0x4;      // Una de las banderas incluidas en la máscara
```

Observe que para el usuario no es importante ni el sitio donde se guardan las banderas, ni el tipo que se usa ni los valores concretos. Realmente, sólo nos interesa que hay una máscara con un determinado nombre y 3 nombres para cada una de las banderas que tiene la máscara.

Para manejarlas, el que crea la biblioteca ofrece al usuario dos funciones:

```
void Reset (int mascara)
{
    estado= estado & ~mascara;
}
void SetFlag (int bandera, int mascara)
{
    estado= estado & ~mascara | bandera;
}
```

donde podríamos haber usado la primera para resolver la segunda.

Con estas funciones puede realizar fácilmente operaciones que activan y desactivan banderas. Por ejemplo:

```
Reset(MascaraTriple); // Deja desactivadas las tres banderas
// ...
SetFlag(Bandera1,MascaraTriple); // Activa la bandera 1 y deja desactivadas 2,3
// ...
SetFlag(Bandera3,MascaraTriple); // Activa la 3 y deja desactivadas 1,2
```

Note que las llamadas a *SetFlag* garantizan que la bandera que queremos activar será la única que quedará activada, independientemente del estado que teníamos antes de la llamada a la función.

Finalmente —“ya metíos en gastos”— imagine que añado una operación más *SetFlag* y mejoro la legibilidad de la función *Reset* anterior:

```
void Reset (int banderaOmaskara)
{
    estado= estado & ~banderaOmaskara;
}
void SetFlag (int bandera)
{
    estado= estado | bandera;
```

El resultado es algo que empieza a recordar a varias de las operaciones que hemos presentado en la figura C.2. Note que la función *SetFlag* nos servirá para banderas independientes, que no están en una máscara o que se pueden activar simultáneamente a otras. Sin embargo, se recomendará usar la función con dos parámetros cuando queramos activar sólo una bandera de una máscara.

## C.4.2 Funciones miembro y manipuladores

En general, se puede decir que el sistema de E/S define dos estrategias para poder enviar órdenes de formateo a un flujo de E/S:

1. Funciones miembro. Se llama a un método del objeto con el operador punto. Por ejemplo, se puede llamar a:

```
cout.fill(' ');
```

para seleccionar el carácter espacio ' ' como el carácter de relleno en **cout**.

2. Un mecanismo que nos permite encadenar esas órdenes de formato con los operadores habituales —<< y >>— de E/S para el flujo. Esta sintaxis facilita la escritura y mejora la legibilidad de estas órdenes.

En algunos casos, tenemos tanto la función miembro como el manipulador disponibles para modificar el formato de E/S (observe que en la figura C.2 sólo se especifica el manipulador en algunos casos). Si usa un manipulador y no se reconoce, es probable que le falte incluir el archivo de cabecera **iomanip**, donde se define.

En la práctica, seguramente lleva usando manipuladores desde que comenzó a programar, ya que **endl** es un manipulador. En este caso, no necesita de la inclusión de **iomanip**, ya que no tiene parámetros.

	Bandera o función miembro	Descripción	Manipulador (si existe)	Notas
Activar/Desactivar banderas general	setf(f)	Activa bandera(s) f y devuelve estado previo	setiosflags(f)	Los identificadores de banderas y máscaras están en std::ios_base, aunque se puede usar std::ios, que es descendiente. Ej: std::ios::adjustfield El ancho de campo se puede usar para la entrada, indicando que sólo se pueden leer n-1 caracteres. Por ejemplo, para leer una cadena C El tipo que corresponde al grupo de banderas es ios::fmtflags. Por ejemplo, se puede hacer cout.setf(ios::fmtflags(0),ios::floatfield) Escriba #include <iomanip> para manipuladores con parámetros. Manipuladores de una bandera en una máscara desactivan el resto de ésta
	setf(f,mask)	Activa bandera f del grupo mask y devuelve estado previo de todas las banderas	resetiosflags(mask)+ +setiosflags(f)	
	unsetf(f)	Desactiva bandera(s) f	resetiosflags(f)	
	flags()	Devuelve todas las banderas		
	flags(f)	Selecciona banderas f como el nuevo estado y devuelve anterior		
Ancho de campo, relleno y ajuste (números, bool, cadena C, etc.)	copyfmt(str)	Copia las banderas desde flujo str		
	width()	Ancho de campo actual (por defecto 0, es decir, lo que se necesite)		
	width(n)	Fija ancho n y devuelve anterior (sólo afecta a E/S siguiente)	setw(n)	
	fill()	Devuelve carácter de relleno actual (por defecto: espacio)		
	fill(c)	Fija c como carácter de relleno	setfill(c)	
Números enteros	máscara adjustfield	left	left	
	right	Ajusta a la derecha	right	
	internal	Signo a la izquierda y valor a la derecha	internal	
	Ninguno	Ajusta a la derecha (por defecto)	resetiosflags(adjustfield)	
	showpos	Escribe el signo a los números positivos	< showpos noshowpos	
Números enteros	uppercase	Usa mayúsculas para números (hexadecimal y científica)	< uppercase nouppercase	
	máscara basefield	oct	oct o setbase(8)	
	dec	Escribe y lee decimal (por defecto)	dec o setbase(10)	
	hex	Escribe y lee hexadecimal	hex o setbase(16)	
	Ninguno	Escribe decimal y lee de acuerdo a los caracteres iniciales	resetiosflags(basefield)	
Punto flotante	showbase	Escribe 0 para octal y 0x para hexadecimal	< showbase noshowbase	
	máscara floatfield	fixed	fixed	
	scientific	Notación científica (precisión= núm. dígitos después de la coma)	scientific	
	Ninguno	La mejor de las dos (por defecto)	resetiosflags(floatfield)	
	precision()	Devuelve precisión actual de punto flotante (por defecto, 6)		
En istream y ostream	precision(p)	Selecciona precisión p y devuelve anterior	setprecision(p)	
	showpoint	Escribe ceros a la derecha del punto decimal		
	bool	boolalpha	< boolalpha noboolalpha	
	flush()	Descarga búfer de salida	flush	
		Inserta salto de línea y descarga búfer	endl	
Otros		Inserta carácter fin de cadena	ends	
		Lee e ignora "espacios blancos"	ws	
	skipws	Salta "espacios blancos" iniciales en cada lectura (por defecto)	< skipws noskipws	
	unitbuf	Descarga búfer de salida tras cada escritura (por defecto para cerr y wcerr)	< unitbuf nounitbuf	

Figura C.2  
CheatSheet para E/S formateada.

## Bibliografía



### Referencias principales

- [1] Garrido, A. *Fundamentos de programación con la STL*. 2016.
- [2] Garrido, A. *Metodología de la Programación: de bits a objetos*. Editorial Universidad de Granada. 2016.

### Referencias secundarias

- [3] Deitel y Deitel, *C++11 for programmers* . Segunda Edición. Prentice Hall. 2013.
- [4] Gonzalez, R. y Wood, R. *Digital Image Processing*. Prentice Hall 2002.
- [5] Garrido, A. *Fundamentos de programación en C++*. Delta publicaciones, 2005.
- [6] Garrido, A, Fernández Valdivia, J. *Abstracción y estructuras de datos en C++*. Delta publicaciones, 2005.
- [7] Donald E. Knuth, *The Art of Computer Programming*, vol. 3 (Sorting and Searching). Addison-Wesley Publishing Company, 2nd edition, 1998.
- [8] Sedgewick, *Algorithms in C++*, Addison Wesley, 1998.
- [9] Stroustrup, B. *El lenguaje de programación C++*. Edición Especial. Addison-Wesley, 2002.
- [10] Stroustrup, B. *The C++ Programming Language, 4th Edition*. Addison-Wesley, 2013.
- [11] Stroustrup, B. *The design and Evolution of C++*. Addison-Wesley, 1994.
- [12] Stroustrup, B. *Programming: Principles and Practice Using C++*. Segunda edición. Addison-Wesley, 2014.

### Referencias electrónicas

- [13] *C++ Reference*. Página web con una referencia bastante completa de los recursos disponibles en el lenguaje C++. <http://www.cplusplus.com/reference/>
- [14] *C++ Reference*. Página web con una referencia bastante completa de los recursos disponibles en el lenguaje C++, incluyendo explicaciones de las aportaciones del último estándar. <http://en.cppreference.com/>
- [15] B. Stroustrup, *C++ Style and Technique FAQ*. [http://www.research.att.com/~bs/bs\\_faq2.html](http://www.research.att.com/~bs/bs_faq2.html).
- [16] GNU, *GNU Make*. <http://www.gnu.org/software/make/manual/make.html>.



# Índice alfabético



## A

acceso aleatorio  
  celdas enlazadas y, 68  
and  
  a nivel de bit, 28  
  matriz booleana, 36  
ar, 14  
archivos cabecera, estilo, 100  
arge, 24  
argv, 24

## B

bandera  
  E/S y, 110  
biblioteca, 13  
  orden de las, 15  
bidireccional, burbuja, 55  
binaria interpolada, búsqueda, 53  
binaria, búsqueda, 52  
bit  
  operador a nivel de, 28  
bool, 4  
  matriz de, 35  
bucle  
  vacío, estilo, 96  
burbuja, ordenación por, 54  
búsqueda, 50  
  binaria con interpolación, 53  
  binaria o dicotómica, 52  
  celdas enlazadas y, 65, 67  
  secuencial, 51  
    garantizada, 52  
  secuencial en un rango, 59

## C

cabecera  
  archivos de, 11  
  localizar archivos de, 20  
camel case, 90  
casting, 5  
cat, 36  
celdas enlazadas, 64

mezcla, 69  
ordenación, 68  
puntero a puntero a celda, 67  
rango de, 71  
cfloat, 2  
char, 1  
class, estilo, 97  
climits, 2  
color  
  imagen de, 30  
  RGB, 30  
coma flotante, 1  
comentario, estilo, 100  
compilación, 12  
  errores de, 13  
  separada, 10  
  unidad de, 12  
compilación separada, estilo, 100  
conectaN, juego, 74  
constante  
  estilo de, 91  
conversión  
  explícita, 5  
  implícita, 4  
CXX, 18  
CXXFLAGS, 18

## D

dec, 3  
declaración, estilo, 99  
define, 12  
delete[], 46  
dependencia  
  make y, 15, 17  
deprecated, 23  
depurar  
  opción de compilador, 18  
desplazamiento de bit, 28  
dicotómica, búsqueda, 52  
dígitos  
  identificador con, 90  
do-while, estilo, 96  
double, 1  
doxygen, 32

## E

E/S  
redireccionamiento de, 22

encapsulación, 38  
  nivel de, 41  
encauzamiento de E/S, 36  
enlazado, 12  
  biblioteca y, 14  
  error de, 13  
enum, 27  
enum class, estilo, 98  
enum, estilo, 98  
espacio blanco  
  estilo y, 92  
esteganografía, 32  
estilo  
  archivos cabecera, 100  
  bucle vacío, 96  
  class, 97  
  comentario, 100  
  compilación separada, 100  
  declaración, 99  
  do-while, 96  
  enum, 98  
  enum class, 98  
  for, 96  
  función global, 97  
  if-else, 94  
  private/protected/public, 98  
  punteros y referencias, 99  
  reglas de, 88  
  struct, 97  
  struct vs class, 98  
  switch, 95  
  while, 96

## F

float, 1  
for, estilo, 96  
función  
  global, estilo, 91, 97  
  miembro, estilo de, 91  
función de biblioteca  
  atoi, 46, 50  
  rand, 46, 50, 103  
  srand, 46, 50, 104  
  strcmp, 49  
  time, 46, 50, 104

## G

g++, 10

gris  
  imagen de, 30

**H**  
hex, 3  
húngara, notación, 91

**I**  
IDE, 88  
identificador  
  estilo de, 89  
IEEE-754/IEC-559, 3  
if-else, estilo, 94  
ifndef/endif, 12  
ifstream, 23  
istream, 23  
imagen  
  de color, 30  
  de grises, 30  
  PGM,PPM, 31  
implícita  
  regla make, 22  
include  
  comillas dobles e, 11  
Inf, 3  
int, 1  
integral,tipo, 1  
interpolación  
  búsqueda binaria con, 53  
iomanip, 111  
ISO8859, 5, 6, 107  
istream, 23

**L**  
LDFLAGS, 18  
LDLIBS, 18  
left, 3  
librería, véase biblioteca  
límites numéricos, 2  
lista  
  de celdas enlazadas, 64  
long double, 1  
long int, 1  
longitud  
  de identificador, 89

**M**  
macro  
  estilo de, 91  
  makefile y, 18  
main, 24  
make, 15  
makefile, 15, 16  
manipulador de E/S, 111  
máscara  
  E/S y, 110  
matriz  
  booleana, 35  
mergesort, 70  
método  
  estilo de, 91  
mezcla  
  de elementos ordenados, 61, 69  
  ordenación por, 70

**N**  
NaN, 3  
new[], 46  
not  
  a nivel de bit, 28  
  matriz booleana, 36  
notación  
  húngara, 91  
numeric\_limits, 2  
números aleatorios, 103  
  pseudoaleatorios, 104

**O**  
objetivo  
  regla make y, 17  
objeto  
  archivos, 13  
oct, 3  
operador  
  nivel de bit, 28  
operadores C++, 108  
optimizar  
  opción de compilador, 18  
or  
  a nivel de bit, 28  
  matriz booleana, 36  
ordenación  
  burbuja, 54  
  celdas enlazadas y, 68  
  por mezcla, 70  
  selección, 49, 69  
  shell, 57  
ordenación  
  estable, 69

**P**  
palabras reservadas, 109  
pascal case, 90  
PGM, formato de imagen, 31  
PHONY, 21  
píxel, 29  
postfijo  
  identificador con, 91  
PPM, formato de imagen, 31  
precompilador, 12  
prefijo  
  identificador con, 91  
private/protected/public, estilo, 98  
pseudoaleatorio, 104  
puntero  
  a función, 56  
puntero nulo  
  listas y, 64  
punteros y referencias, estilo, 99  
punto flotante, 2

**R**  
rand, 46, 50, 103  
rango  
  ordenación de un, 59  
  secuencia y, 58, 71  
  transformación de un, 60  
redireccionamiento  
  de E/S, 22

redireccionamiento de E/S, 36  
regla  
  de makefile, 17  
  ficticia (all), 19  
  implícita, 21, 22, 42  
reservadas  
  palabras, 109  
RGB, 30  
right, 3

**S**  
sangrado, 93  
secuencial  
  búsqueda, 51  
  garantizada, 52  
  búsqueda en rango, 59  
selección, ordenación por, 49, 69  
semilla, 104  
setfill, 3  
setw, 3  
shellsort, 57  
short int, 1  
signed, 1  
signed char, 5  
sizeof, 1  
sobrecarga  
  de funciones, 23  
srand, 46, 50, 104  
strcmp, 49  
struct vs class, estilo, 98  
struct, estilo, 97  
switch, estilo, 95

**T**  
tar, 20  
time, 46, 50, 104  
tipo de dato  
  estilo de, 91  
transformación de un rango, 60

**U**  
unidad de compilación, 12  
union, 7  
unsigned, 1  
unsigned char, 5  
UTF-8, 6, 108

**V**  
variable  
  local  
    estilo de, 91  
vector  
  dinámico, 46  
  reservados vs usados, 47

**W**  
while, estilo, 96  
Windows-1252, 108

**X**  
xor  
  a nivel de bit, 28