

Prácticas con C++

Metodología de la Programación

(2^a ed.)

Antonio Garrido Carrillo





1^a edición 2016

2^a edición 2017

© ANTONIO GARRIDO CARRILLO

© UNIVERSIDAD DE GRANADA

PRÁCTICAS CON C++: Metodología de la Programación (2^a ed.)

ISBN: 978-84-338-6032-3.

Edita: Editorial Universidad de Granada.

Campus Universitario de Cartuja.

Granada. 2017.

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



1	Tipos aritméticos	1
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
2	Gestión de proyectos: make	9
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

3	Esteganografía: texto	27
3.1	Introducción	27
3.1.1	Tipo enumerado	27
3.1.2	Operadores a nivel de bit	28
3.2	Imágenes	29
3.2.1	Niveles de gris y color	30
3.2.2	Funciones de E/S de imágenes	31
3.3	Ocultar/Revelar un mensaje	32
3.4	Desarrollo de la práctica	32
3.4.1	Módulo codificar	33
3.4.2	Programas	33
3.4.3	Práctica a entregar	34
4	Matriz de booleanos	35
4.1	Introducción	35
4.1.1	Condiciones de desarrollo	35
4.2	Problema a resolver	35
4.2.1	Ejemplos de ejecución	36
4.2.2	Formato de archivos	37
4.3	Diseño propuesto	37
4.3.1	Módulo <i>MatrizBit</i>	38
4.3.2	Comprobando la abstracción	41
4.4	Práctica a entregar	42
5	Algoritmos con vectores	45
5.1	Introducción	45
5.1.1	Objetivos	45
5.1.2	Condiciones de desarrollo	45
5.2	Vectores en memoria dinámica	46
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	59
6	Celdas enlazadas	63
6.1	Introducción	63
6.1.1	Objetivos	63
6.1.2	Condiciones de desarrollo	63
6.2	Lista de celdas enlazadas	64
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
7	Reversi	73
7.1	Introducción	73
7.1.1	El juego <i>Otelo/Reversi</i>	73
7.2	Problema a resolver	75
7.2.1	Ejemplo de ejecución	75

7.3	Diseño propuesto: versión 1	78
7.3.1	Interfaces e implementación	79
7.3.2	Programa de la versión 1	80
7.4	Modificación del programa: versión 2	81
7.4.1	Cambios internos a un módulo	81
7.4.2	Cambios en la interfaz de un módulo	81
7.4.3	Ampliar la funcionalidad del programa	83
7.5	Práctica a entregar	86
7.5.1	Versión extra voluntaria	87
A	Gestión de proyectos: CMake	89
A.1	Introducción	89
A.1.1	CMake como <i>meta-generador</i>	90
A.1.2	Otras herramientas relacionadas	90
A.2	Uso básico de CMake	90
A.2.1	Separando fuentes de binarios	90
A.2.2	Un proyecto con varios archivos fuente	92
A.2.3	Variables, la caché y otras opciones	94
A.2.4	Un proyecto con varios directorios y bibliotecas	96
A.3	El lenguaje de CMake	99
A.3.1	Variables	99
A.3.2	Ámbito de las variables	101
A.3.3	Condicionales	102
A.3.4	Bucles	103
A.3.5	Macros y funciones	104
A.4	Módulos	104
A.4.1	La orden <i>find_package</i>	105
A.5	CTest	105
A.5.1	Incorporar tests al proyecto	106
A.5.2	Ejecución de tests: <i>ctest</i>	107
B	Entornos integrados de desarrollo	109
B.1	Introducción	109
B.1.1	Utilidades que ofrece un IDE	109
B.2	El entorno de desarrollo QtCreator	110
B.2.1	QtCreator y QMake/CMake	110
B.2.2	Crear un proyecto	111
B.2.3	Introducción a las opciones del entorno	116
B.3	Compilación y depuración con QtCreator	119
B.3.1	Depuración	120
C	Git: fundamentos	123
C.1	Introducción	123
C.1.1	Control de versiones con Git	123
C.1.2	El concepto de versión	124
C.2	El repositorio local: conceptos básicos	125
C.2.1	Áreas de almacenamiento	125
C.2.2	Estado de los archivos	127
C.2.3	Configuración	131
C.3	El repositorio local: diferencias	133
C.3.1	Tres áreas de almacenamiento, dos bloques de diferencias	133
C.3.2	Mostrando diferencias	137
C.3.3	Historial del repositorio	141

C.4	Repositorios remotos	145
C.4.1	Protocolos del servidor	146
C.4.2	El repositorio <i>origin</i>	147
C.4.3	Avanzando con el repositorio local	148
C.4.4	Subiendo nuestros cambios	149
C.4.5	Descargando una nueva versión	150
D	Guía de Estilo	155
D.1	Introducción	155
D.2	Indicaciones generales	155
D.2.1	Énfasis automático	156
D.2.2	Reglas de estilo	156
D.3	Guía de estilo	156
D.3.1	Identificadores	157
D.3.2	Estructura del código	160
D.3.3	Consideraciones adicionales	166
E	Generación de números aleatorios	171
E.1	Introducción	171
E.2	El problema	171
E.2.1	Números pseudoaleatorios	172
E.3	Transformación del intervalo	173
E.3.1	Operación módulo	173
E.3.2	Normalizar a U(0,1)	173
F	Tablas	175
F.1	Tabla ASCII	175
F.2	Operadores C++	176
F.3	Palabras reservadas de C89, C99, C11, C++ y C++11	177
F.4	Manipuladores y funciones miembro para E/S formateada	177
F.4.1	Banderas y máscaras	178
F.4.2	Funciones miembro y manipuladores	179
F.5	Referencia de operaciones con GIT	179
Bibliografía		183
Referencias electrónicas		183
Índice alfabético		185

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 consultarlo y revisarlo 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.

Además, el documento incluye apéndices con información relevante para realizar los guiones. En concreto, algunos de los apéndices le permitirán 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. En 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.
- Cómo se resuelve 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. Aunque la biblioteca de C++ contiene una serie de tipos y un diseño muy potente para simular distribuciones de probabilidad, en este apéndice sólo introducimos las utilidades que ya hay disponibles con las funciones de C.

- *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.

Por otro lado, el cuaderno de prácticas también está diseñado para que el estudiante pueda enfrentarse a futuros proyectos de una forma más eficaz. Para ello, aun sin ser fundamental para el desarrollo del curso, si presentan una serie de apéndices que pueden ser de especial utilidad para aquellos alumnos que quieran dedicar un esfuerzo adicional a complementar su formación. En concreto, se abordan:

- Una herramienta de *gestión de proyectos*: *cmake*. Siendo de mayor nivel que *make*, permite crear soluciones más fácilmente para proyectos con una modularización complicada.
- Una introducción a los *IDEs*. En concreto, se presenta *QtCreator*. La intención no es tanto estudiar el entorno en su totalidad —que podría necesitar de muchos capítulos— sino presentar lo básico para que el estudiante pueda usarlo. La simplicidad con la que manejar proyectos sencillos en consola debería animar al estudiante a usarlo y con ello facilitarle el desarrollo de proyectos más complejos.
- Una introducción a una herramienta de *control de versiones*: *git*. Sólo se presenta la parte más básica, la que permite empezar a usarlo en pequeños proyectos. Con estos conocimientos, probablemente le sea fácil adaptarse a algún proyecto que requiera de un equipo de varios programadores.

Con estos apéndices, de alguna forma adicionales al núcleo fundamental del curso, dispondrá de las bases necesarias para perder el miedo a desarrollos más complejos, especialmente en cursos posteriores donde esta mejora de la productividad puede reducir sensiblemente el tiempo necesario para resolver esos proyectos.

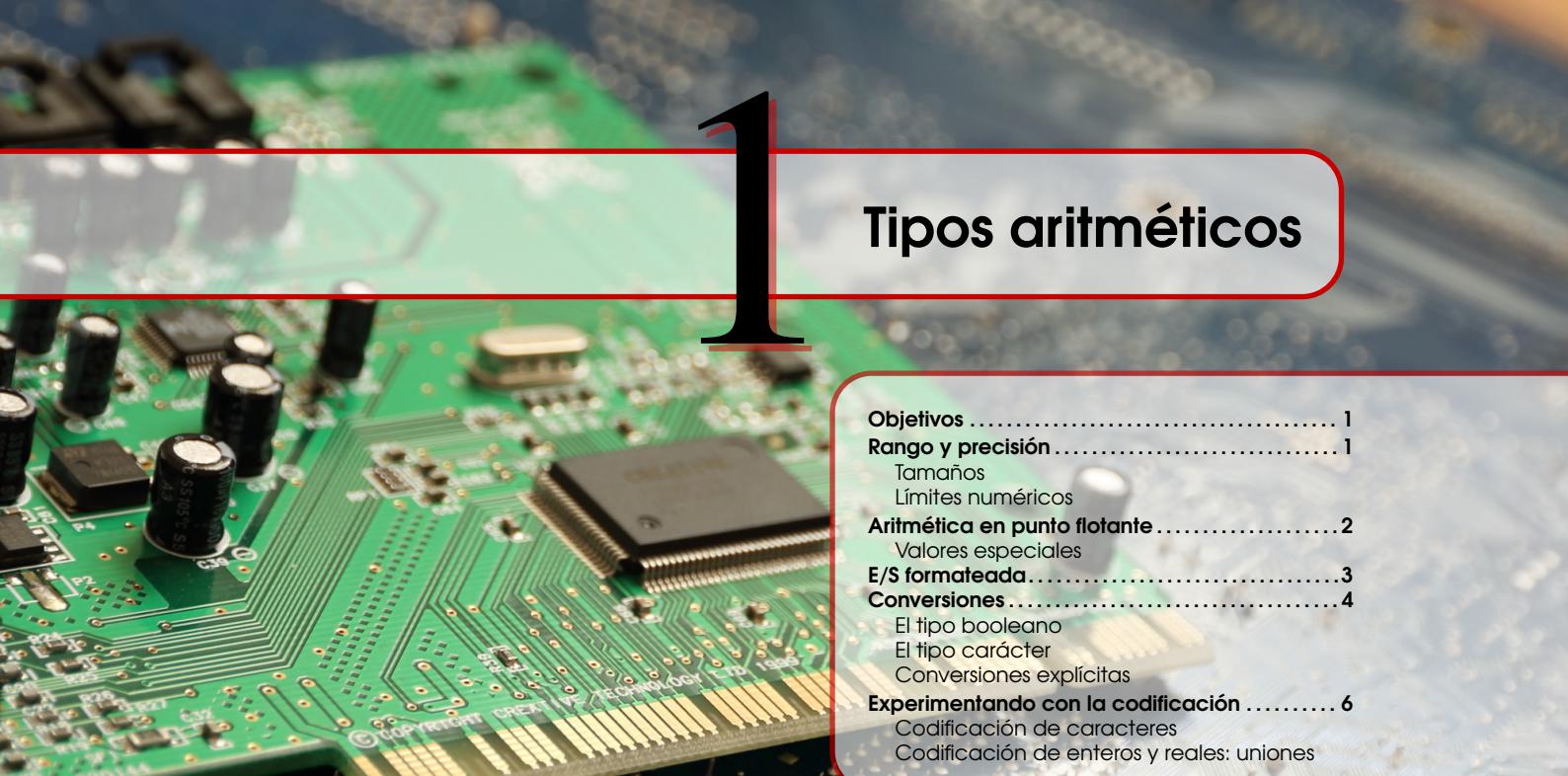
Agradecimientos

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...

A. Garrido.
Febrero de 2017.

1

Tipos aritméticos



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 tamaño¹ de dicho tipo.

¹El 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-7541. 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 ∞ (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 algunos² 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:

²Puede 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 F.4 (página 177) 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**³.

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;
}

```

³Véase la tabla extendida —codificación ISO-8859-15— en la figura F.1, página 175.

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 F.2, página 176). 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 seguros⁴, 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 F.1, página 175.

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 F.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 contenido⁵, 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.

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

⁵Puede 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: make

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 bibliotecas¹. 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

¹En 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 modifiquemos. 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 que será necesario:

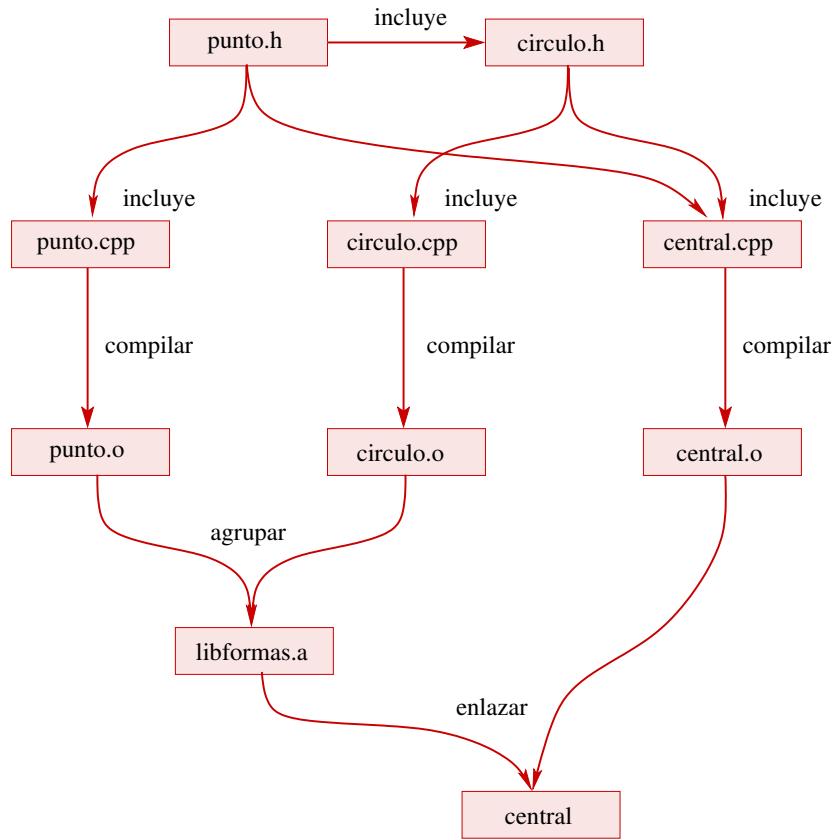


Figura 2.1
Módulos y relaciones para el programa *central*.

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`² (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 '\$'. Existen 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.

²Si 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.

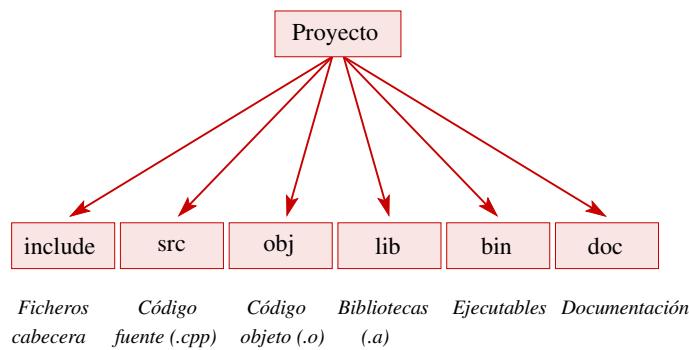


Figura 2.2
Posibles directorios en un proyecto.

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 : include/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:

```
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:

```
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:

```
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 regenerarlo, 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`[18]—. 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 lanzarían. 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 antes³, 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 condena 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”⁴. 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.

³En 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.

⁴Del 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”⁵, 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]`.

⁵Consulte 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(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(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 guión 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:



```
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	
Imágenes	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 puntero¹ 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:

¹En 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.

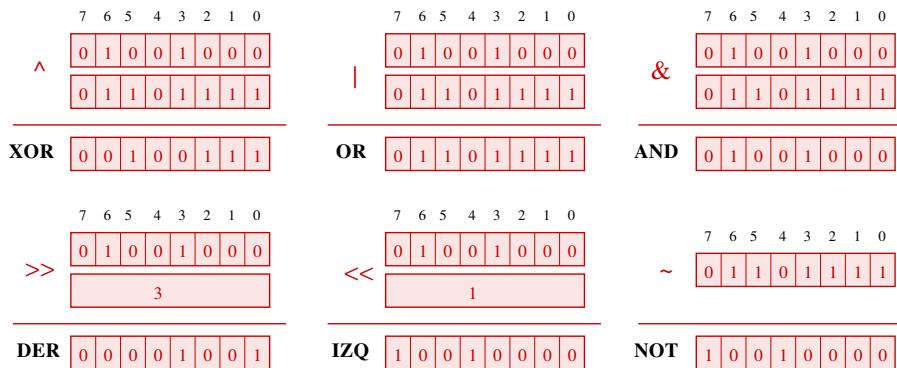


Figura 3.1
Ejemplos de operadores a nivel de bit.

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.

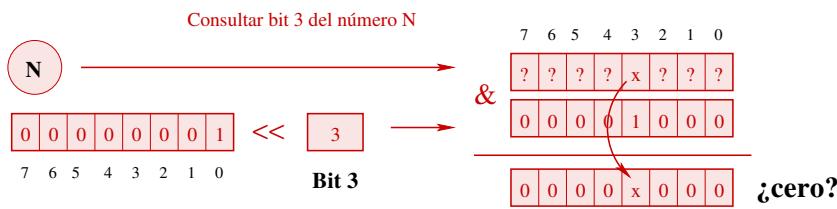


Figura 3.2
Consulta del valor de un bit.

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.

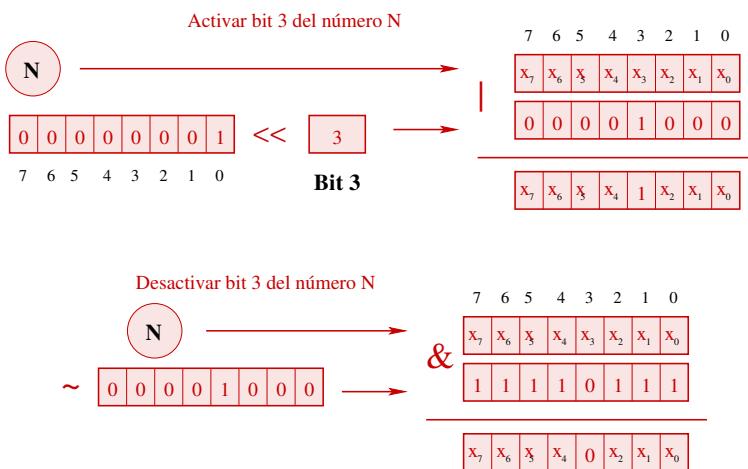


Figura 3.3
Modificación del valor de un bit.

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.

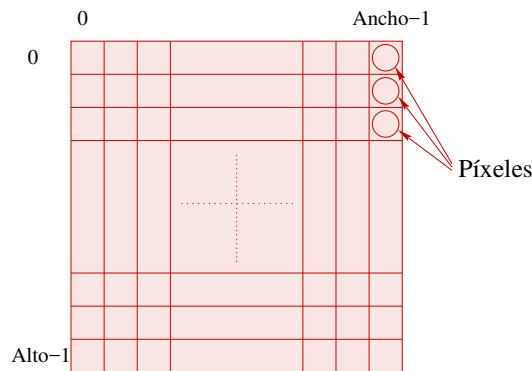


Figura 3.4
Imagen como estructura bidimensional de píxeles.

- 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 byte²(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.

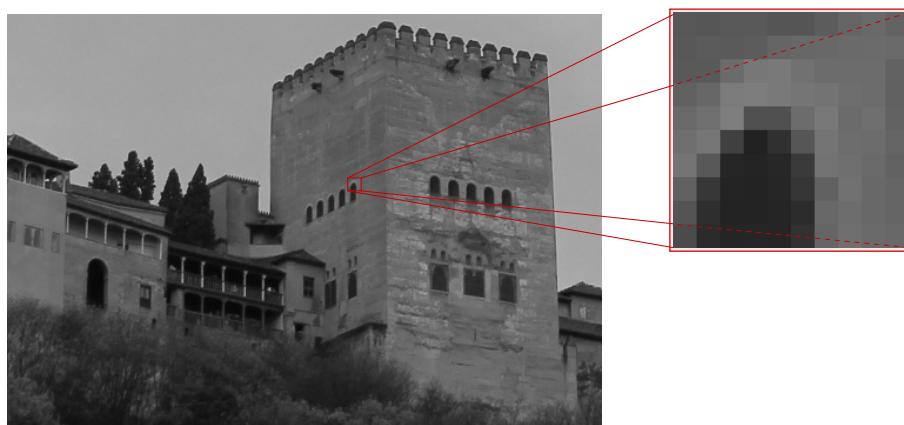


Figura 3.5
Imagen de niveles de gris.

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 azul³.

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

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

³Red, 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.

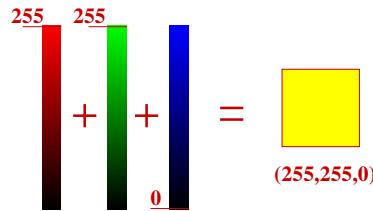


Figura 3.6
Color como mezcla RGB.

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 significativo⁴, 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.

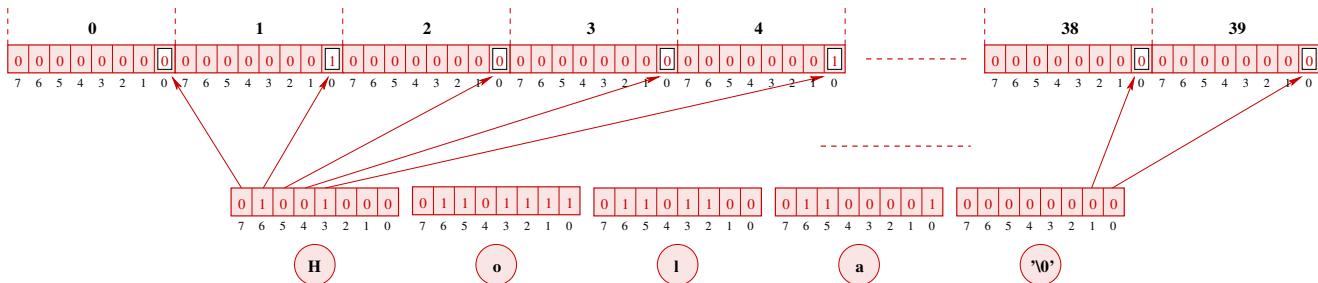


Figura 3.7
Inserción de un mensaje en el bit menos significativo.

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.

⁴El 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:

```
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 × 1.000 de grises ocuparía el 100% de ese espacio, mientras que una 1.000 × 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 × C AND N_F × C**: obtiene una nueva matriz $R_{F \times C}$ con valores $r_{ij} = m_{ij} \&& n_{ij}$.
 - **M_F × C OR N_F × 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:

```

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:

```

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 0 1 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.

Práctica: 4

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.

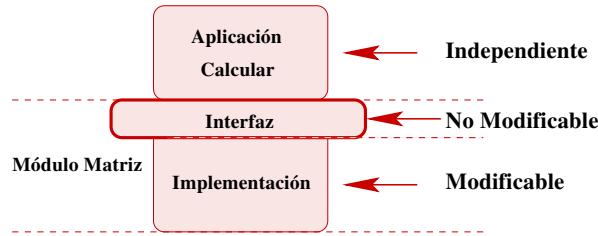


Figura 4.1
Módulo *Matriz* como interfaz más implementación.

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 C¹.

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 operación².

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.

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

²En el material adjunto a este guión encontrará la especificación de cada operación.

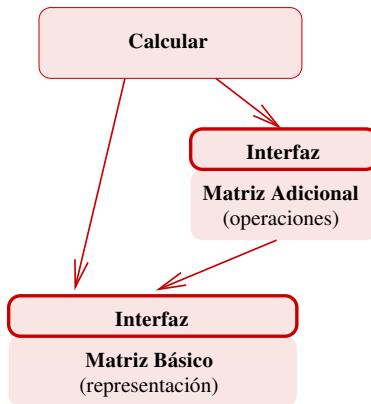


Figura 4.2
Encapsular la representación básica de *Matrix*.

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 *MatrixBit*.

Representaciones del tipo *MatrixBit*

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

1. 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:

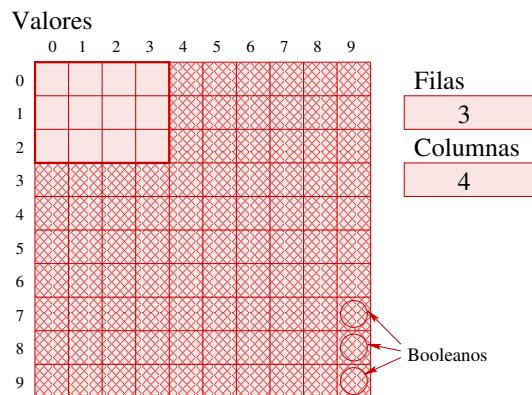


Figura 4.3
Representación basada en una matriz bidimensional.

- Una matriz de tamaño 10×10 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 3×4 .

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.

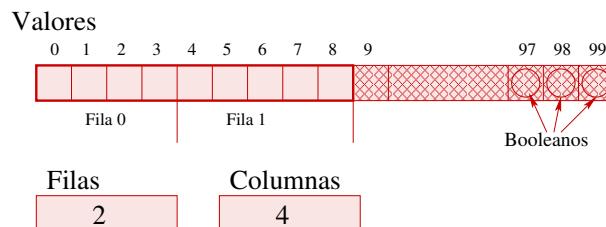


Figura 4.4
Representación basada en un vector.

En este caso, hemos cambiado la matriz de tamaño 10×10 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 11×2 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 2×4 .

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

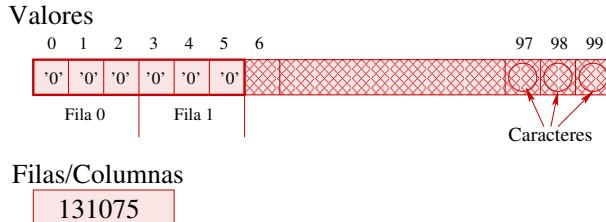


Figura 4.5

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 2×3 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 las 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 2×3 .

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 desperdiando 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 2×3 elementos, usando los 6 bits menos significativos del primer entero de un vector.

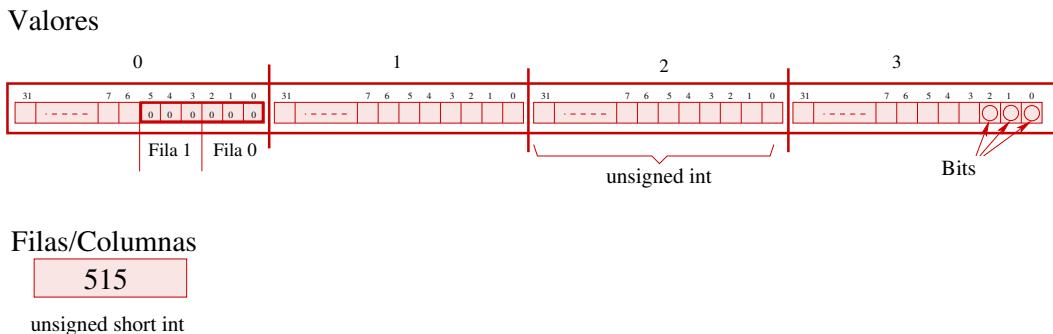


Figura 4.6

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 posiciones³.
- 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.

³Suponemos que un entero tiene 32 bits. Se podría calcular como $8 * \text{sizeof}(\text{unsigned int})$.

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:

```
Consola
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 eficiencia⁴.

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:

⁴De 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.

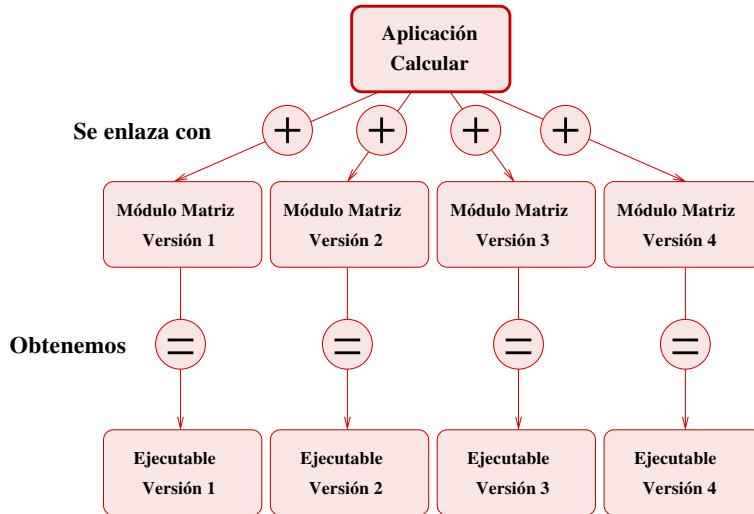


Figura 4.7
Distintas implementaciones del tipo *Matriz*.

- 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, sitúese 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 with a red header bar labeled "Consola". Below the header, there is a scrollable text area containing the following commands:

```
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:

```
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.

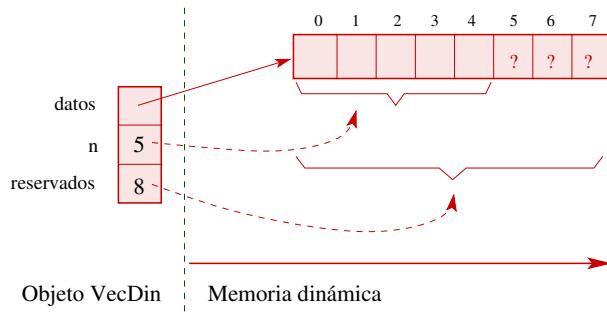


Figura 5.1
Representación de un objeto *VecDin* con cinco datos.

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. El resultado es que se reserva un bloque mayor de memoria y, por tanto, el valor de *n* y *reservados* valdrán lo mismo: el nuevo tamaño.

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 cambiar los tamaños de *n* y *reservados*. Pedir un tamaño menor no afecta a la memoria reservada, sólo cambia el valor de *n*

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 usados 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.

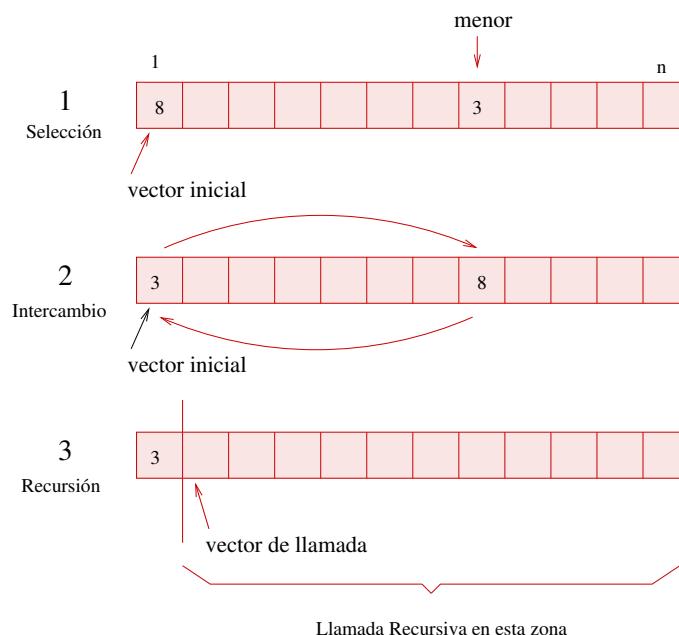


Figura 5.2
Algoritmo recursivo de ordenación por selección.

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

```
Consola
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:

```
Consola
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:

```
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`.

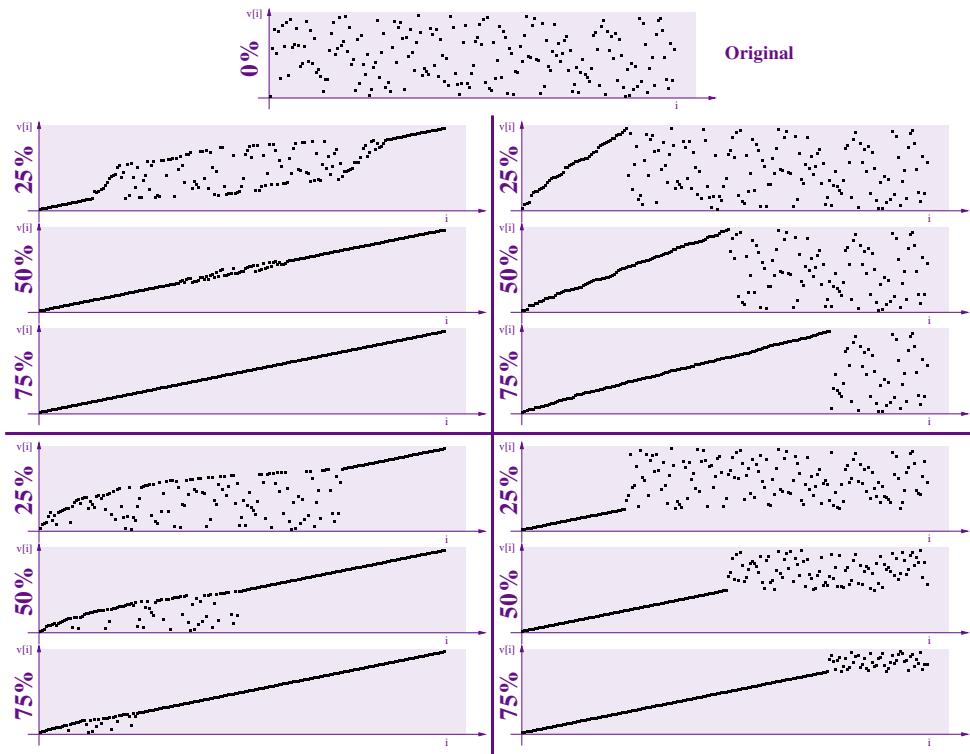


Figura 5.3
Ordenación al 25%, 50%, 75% de algoritmos clásicos.

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.

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 [8] o Sedgewick [9]— para un estudio más formal. En principio, nos limitaremos a decir que son algoritmos simples e ineficientes.

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—.

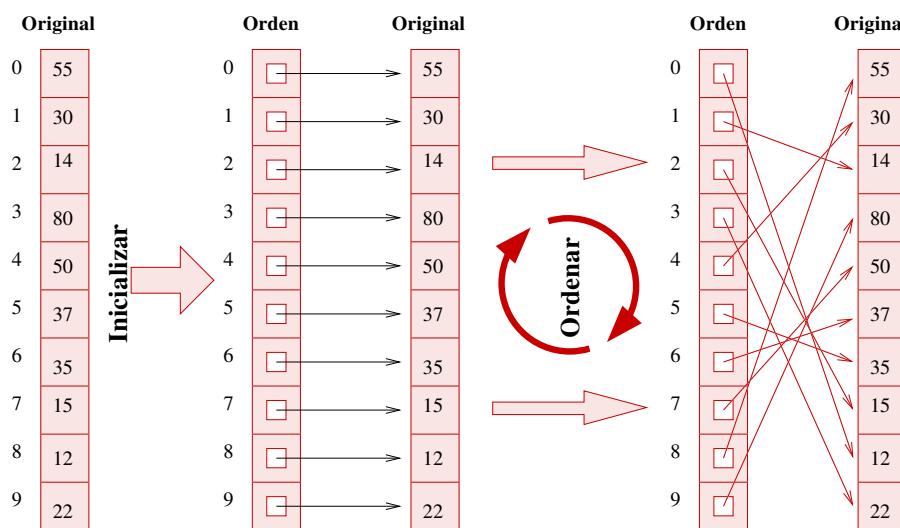


Figura 5.4
Ordenación indirecta con punteros.

Ejercicio 5.22 — Ordenar apuntadores.

- En este ejercicio deberá realizar dos tareas:
1. 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.
 2. 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 3.

Finalmente, suponga que dos elementos son idénticos y consecutivos en el vector de generados. ¿tendrán el mismo valor de salto? Rzone 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 parámetro¹ 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)
{
    return r-l;
}
```

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-l : (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:

¹En 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++.

Consola

```
prompt> ./ordenar_cadenas cadenas.txt
Original:

Hola
Adiós
Prueba
prueba, aunque con minúscula
Mayúscola
minúscula

Resultado:

Adiós
Hola
Mayúscola
Prueba
minúscula
prueba, aunque con minúscula
```

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.

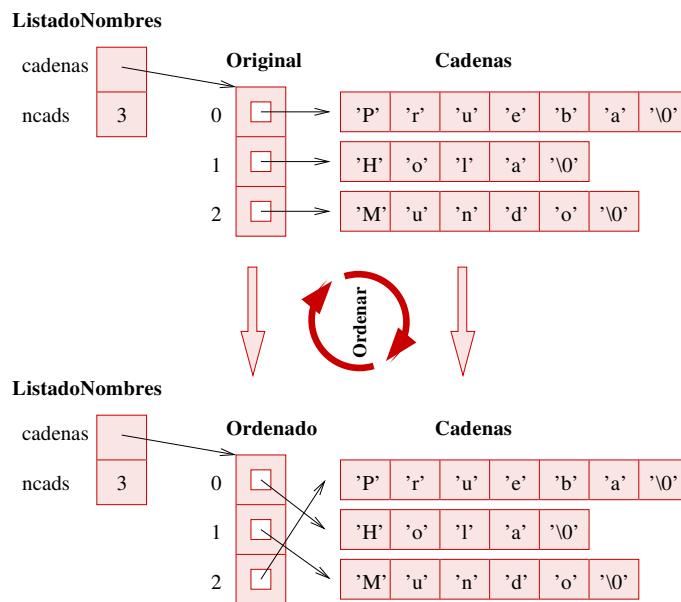


Figura 5.5
Ordenación de cadenas-C

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 manejálos sin ninguna referencia al contenedor inicial —el vector `vec` o el tamaño n — al que corresponden².

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;
```

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:

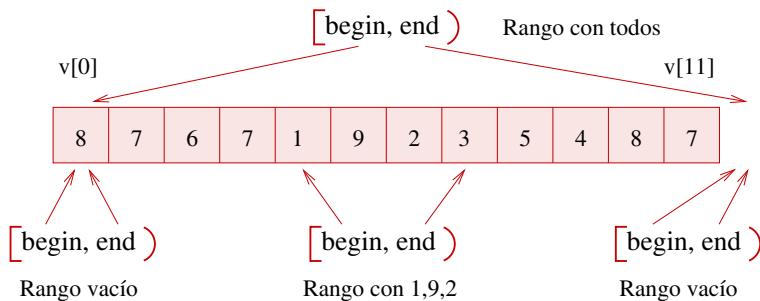


Figura 5.6
Ejemplos de rangos en un vector.

```
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.

²Más adelante estudiaremos la generalización de estos rangos a otros contenedores y comprenderemos cómo podemos usar distintas formas de recorrerlos simplemente con dos “apuntadores”.

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:



```
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.

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 ineficiente 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 `OrdenarGnomo` —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:



```
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 eficiencia³.

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:

```
Consola
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.`

³Serán 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	
Lista 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.

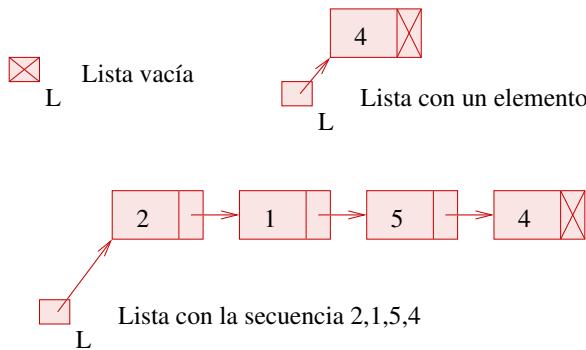


Figura 6.1
Secuencias de datos con celdas enlazadas.

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*.

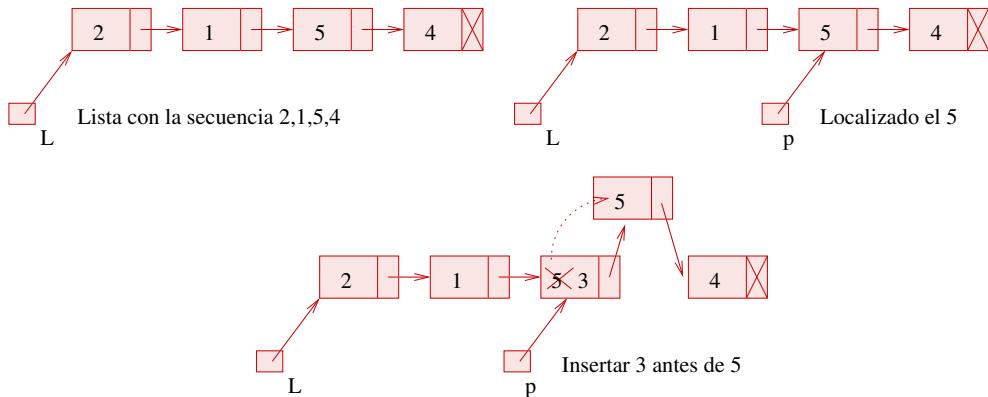


Figura 6.2

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; // Sobreescribimos 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

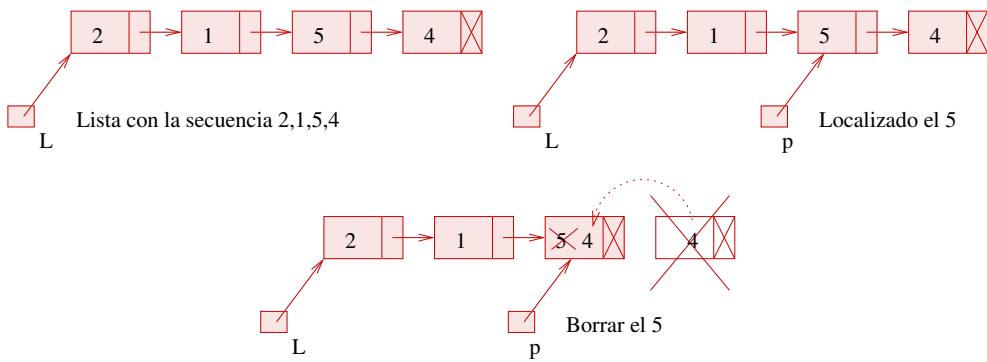


Figura 6.3

Localizar un elemento y borrarlo. Versión no recomendada.

asignación entre celdas¹. 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.

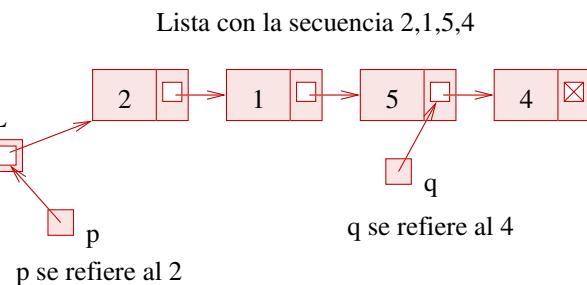


Figura 6.4

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.

¹Aunque 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:

Consola

```
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:

Consola

```
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, añada 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}
```

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 inestable².

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

²Podríamos 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}
Lista1 ordenada: {2,2,2,3,3,4,4,5,5,6,7,7,8,9}
Lista2 ordenada: {1,1,1,2,2,3,5,5,8,8,8,9,9,9,9}
Lista mezcla: {1,1,1,2,2,2,2,3,3,3,4,4,5,5,5,6,7,7,7,8,8,8,9,9,9,9,9}
```

Tenga en cuenta que el algoritmo de mezcla deja las dos listas vacías, por lo que puede aprovechar el identificador *lista1* para guardar la nueva lista ordenada y dejar *lista2* 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 elementos³.
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}
Lista1 ordenada (mergesort): {1,2,3,4,5,5,6,7,7,8,8,8,8,8,8}
Lista2 ordenada (selección): {1,1,2,3,3,4,5,5,6,6,6,7,8,8,9}
Lista mezcla: {1,1,1,2,2,3,3,3,4,4,5,5,5,6,6,6,6,7,7,7,8,8,8,8,8,8,9}
```

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 (1) {
        Celda* acarreo = l;
        l = l->sig;
```

³También 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}
List1 ordenada (mergesort): {1,2,2,2,2,3,3,4,6,7,7,7,8,8,9}
List2 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 59) 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.

Práctica: 6

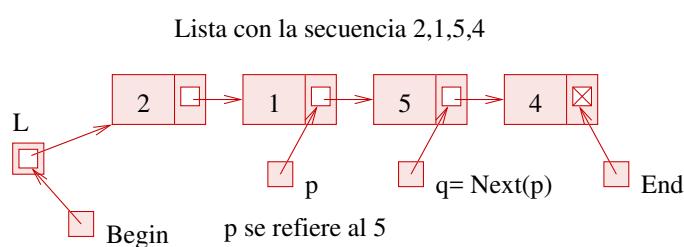


Figura 6.5

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.

7 Reversi



Introducción.....	73
El juego <i>Otelo/Reversi</i>	
Problema a resolver.....	75
Ejemplo de ejecución	
Diseño propuesto: versión 1	78
Interfaces e implementación	
Programa de la versión 1	
Modificación del programa: versión 2	81
Cambios internos a un módulo	
Cambios en la interfaz de un módulo	
Ampliar la funcionalidad del programa	
Práctica a entregar.....	86
Versión extra voluntaria	

7.1 Introducción

En esta práctica se propone crear un programa para jugar al juego *Reversi*, también conocido como *Otelo*. La práctica consistirá en dos versiones del juego —dos soluciones similares— que tienen como objetivo:

- Facilitar la primera implementación para un programador con poca experiencia y mostrar de una forma simplificada el diseño propuesto.
- Mostrar la relación del diseño inicial con la evolución de un proyecto software, en concreto cuando hay que realizar modificaciones y ampliaciones.

La solución que se pide en la práctica está relacionada con la creación de nuevos tipos de datos, especialmente con el encapsulamiento de la representación, eliminando posibles diseños que pudieran incluir otros conceptos de la programación dirigida a objetos, como la herencia o el polimorfismo. Por tanto, la solución deberá estar basada en la creación de nuevas clases que podrán incluir parte privada —**private**— y pública —**public**— para diferenciar la representación de la interfaz.

La primera solución deberá contener tipos de datos simples, en concreto, se deberá evitar el uso de memoria dinámica. En C++, esta solución podrá implementarse de una forma mucho más simplificada, puesto que no será necesario que el usuario implemente la gestión de ningún recurso. El resultado es que en esta primera solución tendremos una versión simple y clara del diseño en módulos. Nos centraremos en el diseño general y en los algoritmos relacionados con el juego.

En la segunda parte, será obligatorio incluir la reserva de memoria dinámica para obligar a que la solución implemente la gestión de reserva y liberación; de esta forma, los recursos necesarios se ajustarán exactamente al tamaño del problema. Para ello, aprovecharemos el encapsulamiento de la primera solución para modificar la representación interna de las clases, incluyendo lo necesario para que las modificaciones se oculten en la parte interna de los módulos. Para demostrar la robustez del diseño general frente a cambios, en esta solución se deberán realizar una serie de ejercicios de modificación y ampliación de la primera. La facilidad para cambiar detalles o añadir funcionalidad mostrará los efectos del encapsulado, así como las ventajas de esta metodología de programación en la evolución de un proyecto software.

Para ambas soluciones, será obligatorio que las soluciones se limiten al uso de punteros, *vectores-C* y *cadenas-C*. El objetivo de esta restricción es obligar a que el estudiante tenga que resolver el problema en base a las herramientas de bajo nivel del lenguaje, generando la necesidad de crear las abstracciones de datos que faciliten la implementación, en concreto, las clases que encapsulen la dificultad de reservar y liberar memoria dinámica.

Por tanto, deberá evitar tipos de la STL¹, como sería el tipo **vector**<> o el tipo **string**.

7.1.1 El juego *Otelo/Reversi*

El juego *Otelo* se juega en un tablero de tamaño 8×8 . Los dos jugadores comparten 64 fichas que tienen dos caras —blanca y negra— de forma que uno juega con blancas y el otro con negras. En la figura 7.1 se presenta un ejemplo de un tablero de estas dimensiones preparado para comenzar la partida, junto con lo que podrían ser las fichas del juego. Se han presentado dos de forma que la cara superior determina que la primera es de un jugador y la segunda de otro. En este tablero se han colocado las cuatro fichas iniciales del juego.

El juego comienza con el jugador que lleva las fichas negras. Poner una ficha en el tablero consiste en elegir una posición de forma que haya fichas del contrario flanqueadas entre dos fichas del jugador: una que ya estaba puesta y la que se coloca en

¹Sin olvidar que en la práctica, si tuviera que resolver este problema sin restricciones, la solución se vería simplificada evitando el uso de memoria dinámica, ya sea porque incluimos memoria automática o porque usamos la STL.

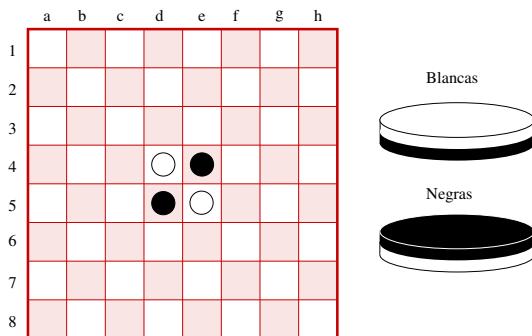


Figura 7.1
Tablero inicial de *Otelo/Reversi*.

ese turno. Por ejemplo, en el tablero de la figura 7.1 el jugador con negras sólo puede poner en 5f y 6e, pues son las dos únicas posiciones donde consigue “encerrar” alguna ficha del contrario entre dos de las suyas. Note que las posiciones que quedan encerradas son de fichas blancas, sin considerar posiciones libres; por tanto, las posiciones válidas serán, necesariamente, adyacentes a alguna posición con ficha blanca.

El efecto de poner una ficha es que todas las fichas del contrario que queden flanqueadas por la que hemos puesto y alguna de las que ya había se dan la vuelta para pasar a ser del jugador que acaba de jugar. Una sola jugada puede dar la vuelta a varias filas/columnas o diagonales siempre que haya en el otro extremo una ficha del mismo color y todas las que quedan encerradas son del contrario.

El turno va cambiando de forma que cada jugador pone una ficha o pasa. Pasar no es una opción del jugador, sino consecuencia de que no haya ninguna posición que permita voltear alguna ficha del contrario. El juego termina cuando no es posible poner más fichas, normalmente cuando el tablero está lleno. El ganador es el que consigue tener más fichas de su color.

En la figura 7.2 se ha presentado un ejemplo de las primeras jugadas en una partida, incluyendo la indicación de la posición seleccionada así como las fichas que se han volteado. Note que tras poner la segunda blanca se cambian dos fichas —aunque no están alineadas— o la tercera blanca, donde se llegan a cambiar tres fichas consecutivas. Tras la última jugada, observe que la posición a1 no es una posición válida para las blancas, pues no es adyacente a una negra que permita flanquear.

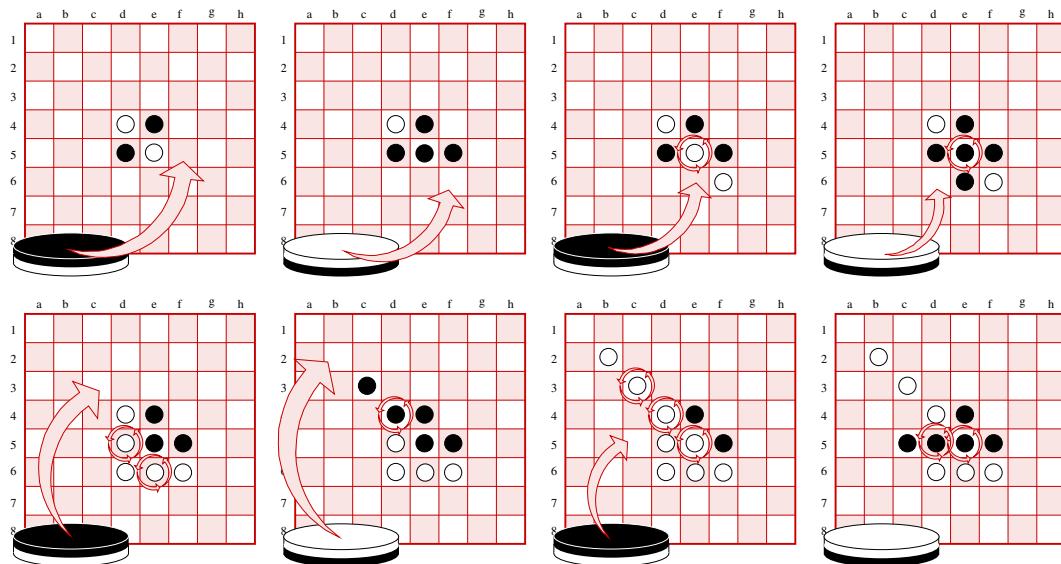


Figura 7.2
Primeras jugadas de una partida.

Por consiguiente, para poder poner una ficha en una posición deberá cumplirse que esa ficha conecta en horizontal, vertical o diagonal con otra del mismo jugador contenido en medio todas y cada una de las posiciones con fichas del oponente. En este caso, decimos que las dos fichas flanquean ese grupo de fichas del contrincante. Además, cuando se pone una ficha, podría flanquear a distintos grupos, pues este flanqueo deberá comprobarse en las 8 direcciones del entorno de la ficha que acabamos de poner.

7.2 Problema a resolver

En este guión se propone crear un programa que nos permita jugar al juego *Otelo*. La interfaz estará basada en el uso de la consola, mostrando el tablero con caracteres *ASCII*. Por ejemplo, a continuación mostramos una situación donde se dibuja un tablero de 8×8 en la situación de inicio de partida:

```

Consola
a b c d e f g h
-----
0| | | | | | | |
1| | | | | | | |
2| | | | | | | |
3| | | x | o | | |
4| | | o | x | | |
5| | | | | | | |
6| | | | | | | |
7| | | | | | | |
-----
```

El juego que se propone comenzará con una versión sencilla con un tablero de tamaño variable relativamente pequeño. Más adelante se propondrán algunos cambios que permiten generalizarlo para probar nuevas variantes.

7.2.1 Ejemplo de ejecución

La mejor forma de entender lo que hará nuestro programa es presentar la dinámica del juego con varios pasos de ejecución. Para comenzar el juego, será necesario conocer el tamaño del tablero y los nombres de los jugadores. El comienzo de la ejecución será:

```

Consola
prompt> ./reversi
Introduzca filas: 6
Introduzca columnas: 6
Introduzca nombre del primer jugador: Fulano
Introduzca nombre del segundo jugador: Mengana
```

El número de filas y columnas es variable, pero aceptaremos únicamente valores entre 4 y 10. Con esa limitación, el tablero no puede tener más de 100 posiciones. Los nombres de los jugadores se necesitan para poder referirse a cada jugador a lo largo de la partida. Estos nombres podrán contener espacios.

Una vez introducidos esos parámetros, el juego iterá preguntando en qué posición se desea colocar la siguiente ficha. Esta posición consiste en un par *letra/número* que determinan columna y fila, respectivamente. Por ejemplo, la primera jugada puede ser *e2*:

```

Consola
a b c d e f
-----
0| | | | | | |
1| | | | | | |
2| | x | o | | |
3| | o | x | | |
4| | | | | | |
5| | | | | | |
-----
```

Turno de jugador 1: (x)
Fulano, escoja una columna (letra a-f) y una fila (0-5): e2

Donde puede ver que el programa ha informado del número de jugador, del carácter asociado en el tablero y del nombre para solicitar una posición. En este caso, esa posición provoca que el juego coloque una ficha y solicite la siguiente jugada:

Consola

```
Fulano, escoja una columna (letra a-f) y una fila (0-5): e2
  a b c d e f
  -----
  0| | | | | |
  1| | | | | |
  2| | x|x|x |
  3| | o|x| |
  4| | | | | |
  5| | | | | |

  -----
Turno de jugador 2: (o)
Mengana, escoja una columna (letra a-f) y una fila (0-5):
```

Aunque esta primera versión es una solución simple, será necesario implementar esta lectura de posición con control de errores. Es decir, si el usuario se equivoca y escribe algo sin sentido o una posición no válida, el programa debe informar de que no es admisible. Por ejemplo, si se equivoca y escribe una posición que no existe:

Consola

```
Turno de jugador 2: (o)
Mengana, escoja una columna (letra a-f) y una fila (0-5): g1
Valores de entrada erróneos
Mengana, escoja una columna (letra a-f) y una fila (0-5):
```

o incluso una posición del tablero pero en la que no tiene sentido colocar la ficha:

Consola

```
Turno de jugador 2: (o)
Mengana, escoja una columna (letra a-f) y una fila (0-5): d4
Posición no válida, no existe flanqueo
Mengana, escoja una columna (letra a-f) y una fila (0-5):
```

el juego responde con un mensaje indicando que esa *posición no es válida*, pues no flanquea fichas del contrincante. Por tanto, el jugador estará obligado a indicar una posición que permita seguir el juego:

Consola

```
Mengana, escoja una columna (letra a-f) y una fila (0-5): c1
  a b c d e f
  -----
  0| | | | | |
  1| | o| | | |
  2| | o|x|x |
  3| | o|x| |
  4| | | | | |
  5| | | | | |

  -----
Turno de jugador 1: (x)
Fulano, escoja una columna (letra a-f) y una fila (0-5):
```

En este proceso de cambio de turno y de solicitud de posición debe tener en cuenta que:

- Es posible que un jugador tenga que pasar. En este caso será el mismo programa quien debe detectarlo e informar de que un jugador pasa para solicitar la siguiente posición al mismo jugador.
- Es posible que ningún jugador pueda poner, normalmente si el tablero está lleno. El juego comprueba no sólo si un jugador debe pasar, sino también que el otro jugador puede poner. En caso de que ninguno de los dos pueda, el juego debe terminar con el resultado.

Por ejemplo, si tenemos la siguiente situación:

Consola

```
a b c d e f
-----
0|o|o|o|o|o| |
1|o|o|o|o|o| |
2|o|o|o|o|x| |
3|o|o|o|x|x| |
4| |x| |x| |
5|x|o|o|o|o|
-----
Turno de jugador 2: (o)
Mengana, escoja una columna (letra a-f) y una fila (0-5): a5
```

el programa responde con:

Consola

```
;Ups! tiene que pasar...
a b c d e f
-----
0|o|o|o|o|o| |
1|o|o|o|o|o| |
2|o|o|o|o|x| |
3|o|o|o|x|x| |
4| |x| |x| |
5|o|o|o|o|o|
-----
Turno de jugador 2: (o)
Mengana, escoja una columna (letra a-f) y una fila (0-5):
```

indicando primero que tiene que pasar y formulando de nuevo la pregunta al mismo jugador.

Si en esta misma partida se vuelve a poner en *f1*, después en *e1* y finalmente el jugador 2 pone en:

Consola

```
Mengana, escoja una columna (letra a-f) y una fila (0-5): e0
a b c d e f
-----
0|o|o|o|o|o| |
1|o|o|o|o|o|o|
2|o|o|o|o|o| |
3|o|o|o|o|o| |
4| |o| |o| |
5|o|o|o|o|o|
-----
Partida finalizada. Ganador: jugador 2
Resultados tras esta partida:
Fulano: 0 ganadas que acumulan 0 puntos
Mengana: 1 ganadas que acumulan 29 puntos
¿Jugar de nuevo(S/N)?:
```

el juego termina porque ningún jugador puede poner una ficha. El programa presenta el resultado de la partida: el ganador ha sido el jugador 2 y los resultados acumulados hasta el momento. Finalmente, solicita si queremos empezar de nuevo con el mismo tablero en la situación inicial. Si decimos que no, terminará presentando los resultados finales:

Consola

```
¿Jugar de nuevo(S/N)?: n
Resultados finales:
Fulano: 0 ganadas que acumulan 0 puntos
Mengana: 1 ganadas que acumulan 29 puntos
0 empatadas
prompt>
```

7.3 Diseño propuesto: versión 1

El objetivo de esta versión es obtener un prototipo rápido, fácil de implementar y que muestre los módulos de la aplicación que desarrollamos así como los algoritmos básicos del juego. La idea más directa es la creación de una clase *Tablero* que se encargará de gestionar el tablero del juego. El problema de cómo funciona el juego se resuelve en esta clase. De alguna forma, incluimos la representación de la información y la forma de jugar.

La simplicidad de la estructura de datos que requiere el tablero podría sugerir que no es necesario incluir más clases. Sin embargo, es interesante separar el problema del almacenamiento de la información del problema de la gestión del juego. Para resolver el problema de la representación, crearemos una clase *Matriz* para gestionar una estructura bidimensional de enteros. Esta solución nos permitirá enfatizar cómo el encapsulamiento nos permite modificar la estructura de datos sin modificar el código que usa la clase.

Además, el programa incluirá un módulo con el programa principal que se ejecutará en la consola. En principio, el programa sería el resultado de unir estos tres módulos en un ejecutable. En la figura 7.3 se presenta un esquema con estos tres modulos. Observe que el programa interacciona con el *Tablero* por medio de su interfaz, mientras que el tablero interacciona con la *Matriz* también a través de su interfaz.

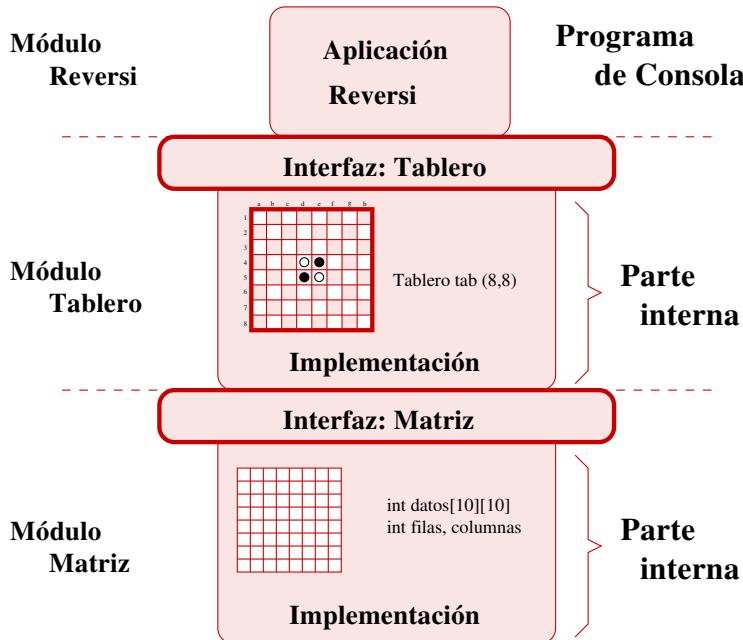


Figura 7.3
Independencia entre el módulo principal y la implementación de matriz.

Por otro lado, no podemos olvidar que en una sesión del juego, el desarrollo de las partidas son el resultado de la interacción entre dos jugadores y el tablero. Los jugadores contienen información sobre su nombre y estadísticas sobre los resultados; además, tienen la responsabilidad de escoger una posición del juego para que la partida avance. En nuestro programa, incluiremos la clase *Jugador* para que se encargue de estas tareas. Como resultado final, el diseño propuesto se basa en las clases:

- Clase *Matriz*. Encapsula la representación de una matriz como una estructura bidimensional de un número de filas y columnas de enteros. La usaremos para representar un tablero. Con ella, el tablero no necesita conocer cómo guardamos el contenido de cada casilla.
- Clase *Tablero*. Un objeto de esta clase almacenará y gestionará el funcionamiento de un tablero. El contenido de cada casilla está resuelto en la clase *Matriz*; en esta clase se incluirán las operaciones propias del juego, como conocer de quién es el turno, modificar el tablero cuando se pone una ficha, etc.
- Clase *Jugador*. Un objeto de esta clase mantendrá los datos relacionados con un jugador, su nombre y estadísticas de juego. También se encargará de interaccionar con el tablero, en concreto, escogerá la posición donde colocar la ficha.

Una representación más completa del juego se puede ver en la figura 7.4, donde también incluimos el módulo *Jugador*. En esta representación, enfatizamos el encapsulamiento y la interacción entre las clases. Por ejemplo, la clase *Tablero* usa la interfaz de la clase *Matriz*, sin acceder a su representación. A su vez, la clase *Jugador* usa la clase *Tablero*, sin acceder a los detalles de implementación.

Una perspectiva muy interesante de la figura 7.4 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*, sin modificar la interfaz del módulo, todo el programa seguirá siendo válido.

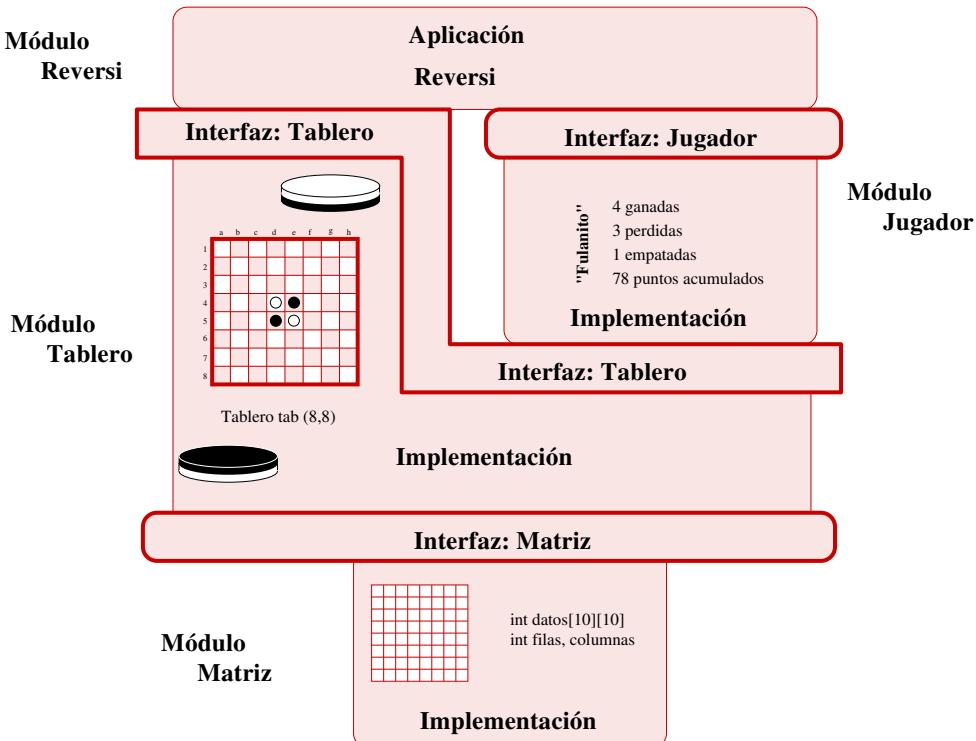


Figura 7.4
Cuatro módulos para implementar el juego *Reversi*.

7.3.1 Interfaces e implementación

Para aclarar las dudas y proponer cierto guión en el desarrollo de esta primera versión, presentamos con más detalle qué representación e interfaces componen las clases anteriores.

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 10×10 . Con esta matriz, podremos representar cualquier tablero que tenga hasta 10 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.

Respecto a las operaciones que ofrece esta clase en la interfaz, se podrían incluir las siguientes:

- Consulta del número de filas.
- Consulta del número de columnas.
- 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 10 por 10 es relevante, porque implica que con un tablero de 6 por 6 vamos a usar sólo 36 de las 100 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 del juego *reversi*. El tablero se crea a partir de sus dimensiones. 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. Cuando se crea, se posicionan las 4 fichas iniciales en el centro y se da el turno al primer 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 contenido de una posición del tablero.
 - 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 si la partida está finalizada.
 - Consulta quién es el ganador. Puede devolver 0 en caso de empate.
 - Consulta la puntuación obtenida en la partida actual. Como precondición, la partida debe estar finalizada.

- Consulta si una posición es válida para colocar la siguiente ficha.
- Consulta si un jugador podría poner ficha o no.
- Operaciones que modifican el tablero:
 - Colocar la ficha en una posición. Note que no es necesario indicar quién inserta. El tablero es el encargado de gestionar los turnos, por lo que la posición 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

Dos objetos de esta clase interaccionarán con un objeto de la clase `Tablero` para implementar el desarrollo de una partida de *Reversi*.

Por un lado, esta clase tiene como objetivo mantener los datos relacionados con 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 `escogePosicion` que realice la elección de una columna. Para nuestro programa, bastará con una función que imprime el tablero en la salida estándar y solicita una posición válida para situar una ficha. La función devuelve como resultado la posición seleccionada, repitiendo la pregunta de la columna mientras no sea correcta. Tenga en cuenta que esta función tiene dos partes:

- Comprobar en cuántas posiciones se podría colocar la ficha. En caso de que haya una sola, basta con devolver ese valor sin necesidad de interactuar con el usuario.
- Si hay más de una posición, será necesario presentar el tablero y solicitar la posición donde colocar la ficha. Este algoritmo usa la E/S de la consola para interactuar con el usuario. Para encapsularlo, escriba esta solución en una función auxiliar `dialogoEscoger` que llamará `escogePosicion`. Siendo auxiliar y no necesaria para el programa, declare la función como método privado de la clase.

La representación deberá incluir 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 quieran.

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 descrito en las secciones anteriores se usarán en un módulo principal que contendrá la función `main` para obtener la funcionalidad descrita en la sección 7.1.1 (página 73). Este módulo principal consistirá básicamente en la petición de parámetros y un bucle que juega repetidamente mientras respondamos afirmativamente a jugar una nueva partida. Cada partida, a su vez, será principalmente un bucle que itera mientras la partida no ha finalizado.

La dificultad de implementar una partida es mínima si tenemos en cuenta que toda la funcionalidad se ha resuelto en las clases anteriores, particularmente en la clase `Tablero`. El bucle pregunta al tablero si ha terminado, si no, tendrá que solicitar al jugador correspondiente al turno qué jugada realiza para, a continuación, modificar el tablero insertando la ficha en dicha posición. Cuando la partida termina, tendrá que indicar a los jugadores que contabilicen los puntos en sus marcadores y preguntar si continúa el juego.

La integración de estos módulos en un programa podría generar distintas situaciones de error. Para simplificar el desarrollo y localizar dichos errores más fácilmente se recomienda insertar código de ayuda a 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).
- Poner una ficha en una posición sólo tiene sentido si permite generar al menos un flanqueo.
- Solicitar que un jugador ponga una ficha sólo tiene sentido si es su turno y tiene al menos una posición donde colocar la ficha.
- 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 diseño del 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.

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 y que éste se ajuste exactamente a las necesidades del programa.

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 7.5 puede ver el tipo de representación que se propone.

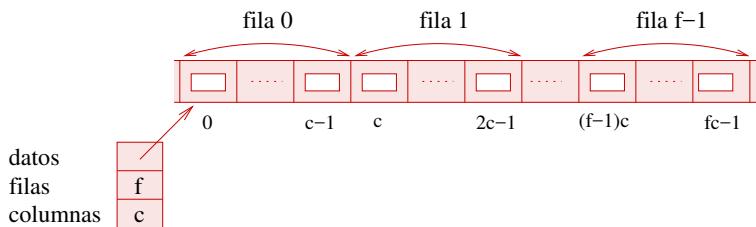


Figura 7.5
Representación de una matriz en memoria dinámica.

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 clase *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 copias, 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 afectan 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 dejarían 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 algún valor 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 *escogePosicion* de la clase *Jugador* para que devuelva un valor booleano que indica si quiere interrumpir el juego. Si lee un carácter '**!**' como columna, directamente ignora el resto de la entrada y devuelve **true** para indicar que no ha seleccionado posición sino la opción de interrumpir la partida.

Marcar como obsoleto

El problema es más complicado cuando queremos cambiar una interfaz y no es posible hacerla compatible. 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* u *obsoleto*. Cualquier software que se desarrolle, debería evitar la funcionalidad marcada como *deprecated* pues en futuras versiones podrían directamente eliminarse². 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. Además, deberá almacenar o recuperar el turno actual.

²Es 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.

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, no sólo en lo que respecta a corrección de errores, sino también en la ampliación de la funcionalidad del programa.

E/S de la configuración y estado del juego

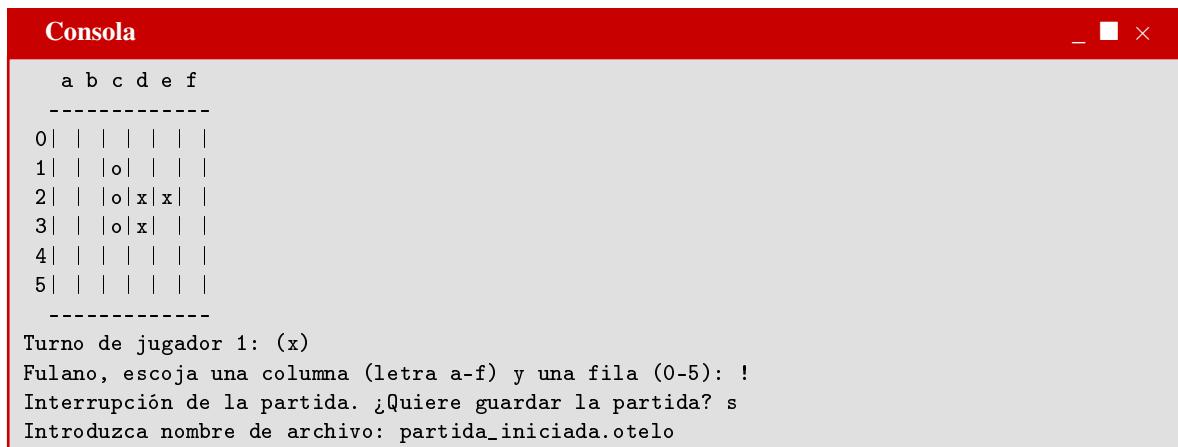
Las operaciones de E/S que se han implementado nos facilitan resolver el problema de almacenar la configuración de una partida e incluso el estado intermedio de una partida. Para añadir esta funcionalidad, se diseña el formato de un fichero de configuración de partida con los siguientes bloques de información:

1. La primera línea es una cadena `"#MP-SUPER-REVERSI-1.0"` que distingue al fichero como configuración del juego *Reversi*.
2. Los parámetros que definen al primer jugador. Note que deberán usarse las funciones de E/S del tipo *Jugador*, tanto para leerlo como para 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 *Guardar* 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 una posición cuando un jugador escoge dónde colocar la ficha. Recuerde que la función que se ha modificado devuelve si queremos interrumpir la partida cuando introducimos el carácter `'!'`. 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:



The screenshot shows a terminal window titled "Consola". The window displays a 6x6 grid representing a game board. The columns are labeled 'a' through 'f' at the top, and the rows are numbered 0 through 5 on the left. The board state is as follows:

```

a b c d e f
-----
0| | | | | |
1| | |o| | |
2| | |o|x|x| |
3| | |o|x| | |
4| | | | | |
5| | | | | |
-----
```

Below the board, the text "Turno de jugador 1: (x)" is displayed. The user is prompted with "Fulano, escoja una columna (letra a-f) y una fila (0-5): !". A message "Interrupción de la partida. ¿Quiere guardar la partida? s" follows, and the user responds with "s". Finally, the user is asked to "Introduzca nombre de archivo: partida_iniciada.otelo".

Observe que hemos introducido `'!'` para que la partida se interrumpa y que el programa solicite si deseamos salvar la partida. En este caso, se introduce un nombre de archivo como destino. Cuando se salva la partida, deberá continuar por donde se dejó:

Consola

```
Introduzca nombre de archivo: partida_iniciada.otelo
  a b c d e f
  -----
0| | | | | |
1| | | o | | |
2| | | o | x | x |
3| | | o | x | |
4| | | | | |
5| | | | | |

-----
Turno de jugador 1: (x)
Fulano, escoja una columna (letra a-f) y una fila (0-5):
```

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: La solución más simple es insertar un nuevo atributo para recordar quién comenzó la partida.*

Ayuda en la selección

Cuando el tablero es complejo la partida se puede hacer muy lenta porque el jugador necesita mucho tiempo para buscar qué alternativas tiene para poner una ficha. Para agilizar este caso, el juego admitirá que un jugador conteste con el carácter ‘?’ de forma que el programa vuelva a preguntar por la posición tras volver a presentar el tablero incluyendo el carácter ‘?’ en cada posible posición válida.

Por ejemplo, en el tablero del ejemplo anterior podría provocar el siguiente diálogo:

Consola

```
Turno de jugador 1: (x)
Fulano, escoja una columna (letra a-f) y una fila (0-5): ?
  a b c d e f
  -----
0| | . | | | |
1| | . | o | | |
2| | . | o | x | x |
3| | . | o | x | |
4| | . | | | |
5| | | | | |

-----
Turno de jugador 1: (x)
Fulano, escoja una columna (letra a-f) y una fila (0-5):
```

Note que parte de la dificultad del juego es descubrir estas posiciones por lo que esto podría considerarse una ayuda para ganar. Para evitar el abuso de esta opción, el juego debe incluir un parámetro adicional con el número de comodines que posee un jugador. El comienzo de la partida será algo similar a:

```
prompt> ./reversi
Introduzca filas: 6
Introduzca columnas: 6
Introduzca número de comodines (0-10): 5
Introduzca nombre del primer jugador: Fulano
Introduzca nombre del segundo jugador: Mengana
```

Ejercicio 7.9 — Ayuda con posiciones. Incluya en el juego que el jugador pueda pedir las posibles posiciones donde colocar una ficha. El formato de las preguntas se cambiará a:

```
Turno de jugador 1: (x)
Fulano, le quedan 2 comodines de 5, escoja una columna (letra a-f) y una fila (0-5): ?
```

Nota: Deberá modificar la clase Jugador, incluyendo las funciones de E/S. Además, deberá incluir un segundo parámetro en la función prettyPrint para indicar si quiere incluir este carácter de ayuda..

Jugador automático

Una modificación interesante es la inclusión de un jugador automático, es decir, que en lugar de solicitar la posición donde colocar la ficha, 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 7.10 — Incluir un jugador automático. Modifique la función `escogePosicion` para que incluya la posibilidad de un jugador automático. Para ello, comprobará si el nombre comienza por '@', en cuyo caso seleccionará cualquier posición válida de forma aleatoria. *Nota: Para realizarlo deberá crear una lista de todas las posibles posiciones y escoger una, por ejemplo, de forma aleatoria.*

Cambiar tablero y estrategia de turnos

Una vez que hemos implementado la versión clásica del juego, es interesante explorar posibles modificaciones que hagan el juego más atractivo. Una forma de probar otra alternativa es cambiar la forma en que se alternan los turnos. En lugar de que un jugador ponga una ficha como máximo se permite que ponga más de una. Por tanto, el juego lo determina las dimensiones del tablero y el número de fichas máxima por turno. Tenga en cuenta que si el jugador ya no puede poner ninguna —tiene que pasar— no importará cuántas ha puesto, el turno cambiará al otro jugador, que dispondrá también de ese máximo.

Para implementar esta estrategia tendríamos que realizar varios cambios en el programa, como preguntar el número de fichas por turno y modificar varias funciones. El problema de esta configuración es que el tablero inicial que hemos presentado hasta ahora no es válido. Por ejemplo, si situamos las 4 posiciones centrales con dos pares de fichas, el primer jugador ganará al poner las dos primeras y eliminar al contrario.

Para resolver el problema rehacemos la parte que corresponde a la configuración del tablero. En concreto, vamos a jugar siempre con un tablero previamente almacenado de forma que la ejecución del programa necesita simplemente leer el nombre del tablero y los parámetros relacionados con los jugadores. Un posible comienzo de partida será, por tanto:

```
prompt> ./reversi
Introduzca tablero inicial: 10x10con3fichas.otelo
Introduzca número de comodines (0-10): 5
Introduzca nombre del primer jugador: Mengana
Introduzca nombre del segundo jugador: Zutano
```

Para incluir esta nueva forma de jugar tendremos que realizar varios cambios en el programa. Los más importantes son en la clase *Tablero*:

- El constructor de la clase *Tablero* también tendrá un parámetro que indica el número máximo de fichas por turno. Este parámetro lo restringiremos en el rango [1,3].
- Será necesario incluir 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.
- Se deberá modificar la E/S del tipo *Tablero* para que incluya la nueva información asociada.
- Se debe modificar la función que pone una ficha en el tablero, pues el turno cambia de otra forma.
- La función *prettyPrint* que informa del estado del tablero debe incluir información sobre la ficha y el máximo de fichas. Por ejemplo:

```
Consola
Turno 2/3 de jugador 1: (x)
Fulano, le quedan 2 comodines de 5, escoja una columna (letra a-f) y una fila (0-5):
```

- Podemos añadir una función *cargar* a la clase *Tablero* para leer un tablero desde un fichero. Este fichero será distinto al de configuración de partida. Para distinguirlo, creamos un nuevo formato que consiste en:
 1. La primera línea es una cadena #MP-TABLERO-REVERSI-1.0 que distingue al fichero como configuración de tablero.
 2. La configuración del tablero. El formato —y la solución— corresponde a la función de lectura del tipo *Tablero*.
 3. Entre los datos que componen la configuración, el fichero puede contener un carácter '%' que indica que lo que sigue es un comentario hasta final de línea.

Ejercicio 7.11 — Modificar estrategia de turnos. Modifique el programa para incluir la funcionalidad que se ha descrito en esta sección. *Nota: Puede descargar algún ejemplo de fichero de configuración para que le sirva de ejemplo y prueba.*

7.5 Práctica a entregar

Debe crear una carpeta —por ejemplo, **reversi**— 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.

El curso tiene como objetivo aprender a usar los archivos «**makefile**», por tanto, será obligatorio y se evaluará la solución basada en la generación del ejecutable a partir de este tipo de archivos con «**make**». Además, no olvide que deberá incluir operaciones que permitan la “*limpieza*” de archivos intermedios generados.

Por otro lado, se recomienda conocer otras herramientas, en concreto, el uso de «**cmake**» y los archivos «**CMakeLists.txt**». Para facilitar que el estudiante pueda probar este tipo de configuración, se admitirán las soluciones en las que la segunda versión se gestione de esta forma³.

En la figura 7.6 se presenta un esquema de cómo quedarán si incluye esta recomendación. En caso de que no quiera usar «**cmake**», la segunda versión tendrá que tener un esquema similar a la primera. Observe que no hay un directorio para bibliotecas, pues el ejecutable se generará directamente a partir de los archivos objeto.

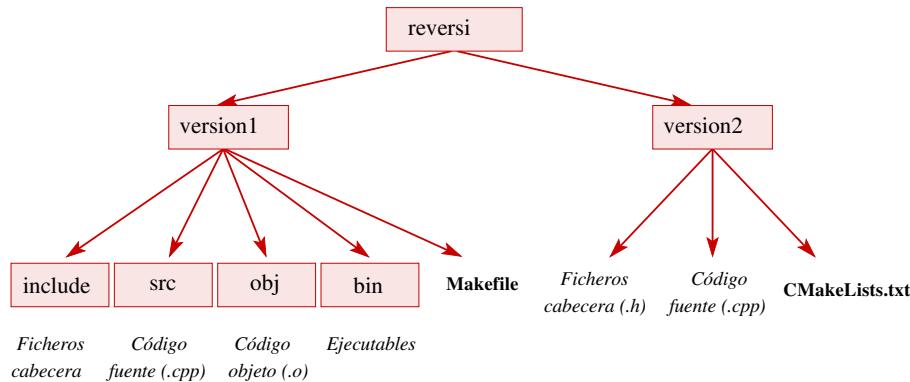


Figura 7.6
Directorios del proyecto *reversi*.

³De hecho, al estar fuera de los objetivos del curso, se facilitará un archivo para que pueda trabajar sin necesidad de dedicar mucho tiempo a «**cmake**».

Recuerde que 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. Si quiere usar «`make`», el archivo `Makefile` y toda la estructura que tenía será válido para realizar las modificaciones para la segunda versión. Si quiere usar «`CMakeLists.txt`», copia todos los archivos a un mismo directorio. Recuerde que esta herramienta facilita la generación de objeto y ejecutables en un directorio aparte.

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, sitúese en la carpeta superior y use la orden `tar` para obtener el archivo resultado. Más concretamente, ejecute lo siguiente:



```
prompt> cd version1
prompt> make clean
prompt> cd ../..
prompt> tar zcvf reversi.tgz reversi
```

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

7.5.1 Versión extra voluntaria

Se propone una tercera versión para el proyecto de este guión práctico. En esta versión se pretende cambiar la interfaz del programa a un entorno gráfico para que la selección de una casilla sea más fácil. No se pretende realizar un cambio radical con una biblioteca que pudiera implementar todos los detalles (como la inclusión de menús, diseño de botones, etc.); en lugar de eso, nos limitaremos a simplificar la selección de una casilla.

De esta forma, el estudiante de primer curso podrá introducirse en el uso de un entorno gráfico con un pequeño esfuerzo que se centre, especialmente, en explotar algunos conceptos del curso como son la modularización o la creación de bibliotecas. Además, podrá aprovechar para conocer algunas herramientas que le pueden ser útiles para cursos más avanzados.

Programa propuesto

La tercera versión corresponde a un proyecto con dos ejecutables. Por un lado tendremos que reestructurar la segunda versión para rehacer el programa `reversi` y por otro se propone la creación de un programa `otelo` que ofrece una interfaz gráfica mínima. Los detalles de cómo crear estos módulos los puede encontrar en la documentación que se puede descargar de la página asociada al curso.

El nuevo programa `otelo` es un ejecutable que se lanza desde la consola con dos parámetros en la lista de argumentos:

- *Un nombre de archivo que contiene la partida.* Este tipo se puede cargar con la función que ya ha implementado en la segunda versión del programa. En los ficheros que puede descargar de la página del curso encontrará varios ejemplos. Revise su contenido para hacerse una idea de qué contienen y, si lo desea, adáptelo al formato que ya ha resuelto en la versión anterior. En cualquier caso, los archivos de partida que haya salvado anteriormente deberían funcionar sin problemas.
- *Un directorio de recursos gráficos y de sonido.* Este directorio es necesario para poder inicializar la ventana gráfica. Consulte la documentación para ver qué puede dar el nombre del directorio que ha descargado de la página del curso.

Por tanto, no tendrá una forma de ejecución que implique la introducción de la configuración en la consola. Por ejemplo, la ejecución sin parámetros generará un mensaje de ayuda:



```
prompt> ./otelo
Uso: otelo/otelo <nombre archivo> <directorio recursos>
```

En la ejecución del programa se va a evitar el uso de la consola en la introducción de datos, aunque seguiremos obteniendo resultados. En concreto:

- Puede pulsar la tecla `'!'` en la ventana del juego. Cuando selecciona esta opción de interrumpir para poder salvar la partida, no se solicita ningún nombre de archivo. En lugar del diálogo de salvar, el programa reescribe el archivo original con el estado actual de la partida. Por tanto, si desea usar varias veces una configuración, copie el archivo cada vez que desea inicializar un enfrentamiento.
- Puede pulsar la tecla `'?'` en la ventana del juego. El resultado es que se muestran las posibles casillas gráficamente, si quedan comodines.
- El programa seguirá manteniendo la salida de resultados por consola. Note que no será necesario introducir datos, sólo obtendrá resultados de cada partida o los resultados finales para conocer las puntuaciones obtenidas.

Directorios del proyecto otelo

Con esta nueva versión, la entrega de la práctica contendrá tres subdirectorios. Recuerde que las dos versiones anteriores son obligatorias. Por tanto, dispondrá de dos ejecutables *reversi* —en la versión 2 y la 3— con la misma funcionalidad, aunque con un diseño modular de los fuentes distinto. El resultado es un árbol de contenidos como el que se presenta en la figura 7.7.

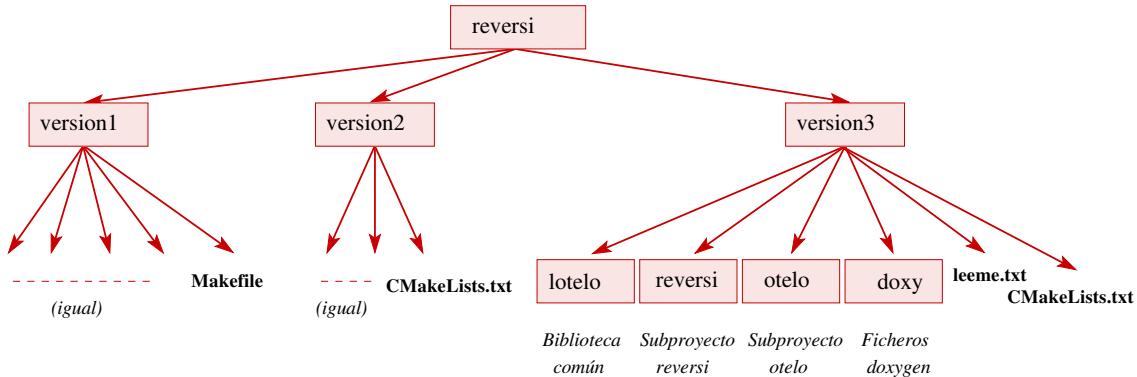


Figura 7.7
Directorios del proyecto *reversi/otero*.

Tenga en cuenta que en esta figura no se presentan los contenidos de los directorios de la versión 3. Para ello, descargue los archivos de la página de la asignatura y revise los directorios y los archivos incluidos.

Consulte el archivo `leeme.txt` para generar la documentación `doxygen` que incluirá más detalles sobre lo que tiene que realizar. En concreto, la página principal describe con más detalle estos contenidos y las tareas para obtener la solución.

Entorno de desarrollo

Se recomienda usar un entorno de desarrollo para esta versión. Además, el esqueleto que se ofrece contiene archivos **CMakeLists.txt** que configuran los módulos y ejecutables. En los archivos descargados tiene prácticamente la solución final, a falta de añadir los nombres de los archivos que forman la biblioteca en el archivo **CMakeLists.txt** del subdirectorio **lotelo**. El resto debería funcionar correctamente.

No es objetivo del curso la discusión sobre gráficos, un entorno de desarrollo completo o la herramienta «CMake». Por ello, si tiene alguna duda, consulte al profesor. Los problemas que debe solucionar deberían centrarse en la modularización de sus soluciones y el uso del módulo «ventana», especialmente diseñado para que pueda obviar la gestión de gráficos⁴.

⁴Una buena solución requiere mucho más trabajo y posiblemente una buena biblioteca de *widgets* que le faciliten completar la interfaz gráfica. Sin embargo, se sale de los objetivos del curso.



Gestión de proyectos: CMake

Introducción.....	89
CMake como <i>meta-generador</i>	
Otras herramientas relacionadas	
Uso básico de CMake	90
Separando fuentes de binarios	
Un proyecto con varios archivos fuente	
Variables, la caché y otras opciones	
Un proyecto con varios directorios y bibliotecas	
El lenguaje de CMake	99
Variables	
Ámbito de las variables	
Condicionales	
Bucles	
Macros y funciones	
Módulos.....	104
La orden <code>find_package</code>	
CTest	105
Incorporar tests al proyecto	
Ejecución de tests: <code>ctest</code>	

A.1 Introducción

En el capítulo 2 se ha estudiado la orden «`make`» y los archivos «`makefile`» como una herramienta de gestión automática de la compilación y enlazado de programas C++. Realmente, este sistema no se ha creado específicamente para este lenguaje, sino que puede usarse para otros lenguajes de programación o incluso otros tipos de archivos.

En general, este tipo de herramientas gestionan automáticamente la construcción de objetivos a partir de fuentes, es decir, automatizan el proceso de forma que la modificación en alguna de las fuentes provoca la regeneración de objetivos. La ventaja de usarlos es que evitamos la tarea de comprobar los cambios que se han realizado y podemos minimizar el tiempo de regeneración puesto que el sistema se limita a regenerar sólo los objetivos necesarios, los que han sido afectados por los cambios.

Los sistemas automáticos de generación de objetivos se basan en procesar un archivo (o archivos) —en el caso de «`make`» el archivo «`makefile`»— que describe la relación entre las fuentes y los objetivos así como la información necesaria para poder regenerarlos.

La orden «`make`» se usa ampliamente y es un buen ejemplo para introducirse en la gestión automática de proyectos. Para ejemplos sencillos es suficiente y resulta muy didáctica. Sin embargo, no es la única, ni tampoco la ideal para todos los casos, especialmente si trabajamos en grandes proyectos o con otros lenguajes.

A continuación listamos algunas alternativas, no tanto para que conozca ningún nombre en particular, sino para que sea consciente de la amplia variabilidad que puede encontrar:

- «`Ninja`». Es similar a «`make`» con menos posibilidades, más simplificado, pero también más rápido.
- «`Scons`». Usa el lenguaje «`Python`» en sus archivos de configuración.
- «`Shake`». Usa el lenguaje «`Haskell`».
- «`Ant`». Muy relacionado con proyectos «`Java`». Es parte de la *Apache Software Foundation*.
- «`Gradle`». En este caso, está relacionado con el lenguaje «`Groovy`», que también trabaja sobre la plataforma «`Java`». Es también parte de la *Apache Software Foundation*.
- etc.

Como puede ver si busca información adicional sobre estos y otros paquetes, la gestión de un proyecto no es un tema baladí, especialmente cuando se trata de un proyecto de gran tamaño. Para este tipo de proyectos, es probable que no use «`make`» para su gestión, sino herramientas de mayor nivel. En muchos casos, podrá gestionarlo de forma transparente desde un entorno de desarrollo integrado —*IDE*— que, además, podría usar alguna de estas herramientas internamente para resolverlo.

En este tema se presenta «`CMake`» como una herramienta de mayor nivel para facilitar el desarrollo especialmente en C/C++. En algunos casos, puede encontrarla algo incómoda en su sintaxis, puesto que tiene su propio lenguaje *script*, pero en muchos casos resulta mucho más recomendable porque:

- Es más simple que escribir complejos archivos «`makefile`».
- Obtiene una solución más portable. El resultado es que puede crear su proyecto pensando en un uso multiplataforma, para compilarlo fácilmente en *Windows*, *Gnu/Linux* o *MacOS*.
- Facilita su uso junto con otras soluciones de desarrollo, incluyendo la generación de archivos de proyecto para entornos de desarrollo como *Visual Studio*, *CodeBlocks* o *XCode*.

Si tiene que realizar un proyecto de cierto tamaño, es recomendable evaluar el uso de este sistema de gestión automática. Estudiar cómo funciona facilitará la comprensión de otros sistemas así como el uso de entornos de desarrollo que se basan en ellos. Por ejemplo, el entorno *QtCreator* puede usar *CMake*, aunque está especialmente diseñado en las soluciones con las librerías *Qt*, que usan una herramienta similar: *qmake*¹.

A.1.1 CMake como meta-generador

«CMake» no funciona como la orden «*make*» analizando una serie de reglas y lanzando las órdenes para regenerar los objetivos. No genera objetivos, sino los archivos generadores de objetivos para distintos sistemas. Por ejemplo, puede procesar los archivos de configuración del proyecto —veremos que son archivos *CMakeLists.txt*— para generar los archivos «*makefile*» que permiten obtener los objetivos finales con la orden «*make*».

Por tanto, lo podríamos calificar como *meta-generador*, un generador de archivos generadores. Además, se puede lanzar la orden «*cmake*» para distintos sistemas de generación de objetivos sin cambiar los archivos de configuración. Esta orden no compilará el programa, sino que generará la solución para que sea otra herramienta la que finalmente gestione la solución.

Por ejemplo, puede trabajar en *Gnu/Linux* generando los archivos «*makefile*» automáticamente para que la orden «*make*» sea la encargada de compilar. El mismo proyecto lo puede llevar a una plataforma *Windows* aunque indicando a «*cmake*» que quiere la solución de *Visual Studio*, o tal vez a *MacOs X* para el que generará un resultado que permita obtener la solución a través del entorno *Xcode* de *Apple*.

A.1.2 Otras herramientas relacionadas

Junto con *CMake* se pueden usar otras herramientas que le ayudarán en otra tareas relacionadas con el desarrollo, concretamente:

- *CPack*. Para ayudar en la tarea de empaquetar el resultado.
- *CTest*. Para ayudar en la tarea de gestionar automáticamente en el desarrollo y ejecución de test.
- *CDash*. Para ayudar en la tarea de reunir, analizar y mostrar los resultados de los tests desde distintas fuentes.

Finalmente, es importante recordar que todas estas herramientas son de código abierto y están desarrolladas por *KitWare* como una solución para el desarrollo de proyectos multiplataforma, como son los casos de *ITK* y *VTK* de la misma empresa.

A.2 Uso básico de CMake

En esta sección revisamos algunos aspectos básicos del lenguaje de *scripting CMake*, evitando gran parte de las órdenes y detalles que incluye; no es nuestro objetivo estudiarlo a fondo, puesto que para los proyectos que desarrollamos, conocer los fundamentos y las órdenes más relevantes es más que suficiente para hacer muy simple la gestión de software con este herramienta².

En esta primera parte vamos a presentar algunos ejemplos y las órdenes más útiles para resolver por completo un proyecto relativamente simple. El lenguaje es bastante intuitivo, por lo que es probable que con los ejemplos le sea suficiente para reproducir otras soluciones en sus propios proyectos. Para aquellos más interesados, se amplía la discusión en las siguientes secciones a fin de poder conocer más detalles y animar a estudiar mejor otras soluciones más complejas.

A.2.1 Separando fuentes de binarios

Lo primero que debemos tener presente para trabajar con *CMake* es que vamos a separar los archivos fuente de los binarios. En los primeros programas es habitual tener los archivos en un mismo directorio, mezclados, porque son pocos archivos y resulta muy cómodo ignorar la dificultad de relacionar archivos en distintas localizaciones. Sin embargo, en un proyecto de mayor tamaño la separación es necesaria. No es algo propio de *CMake*, sino de muchos esquemas de trabajo con otras herramientas. En la figura A.1 se presenta esta idea gráficamente.

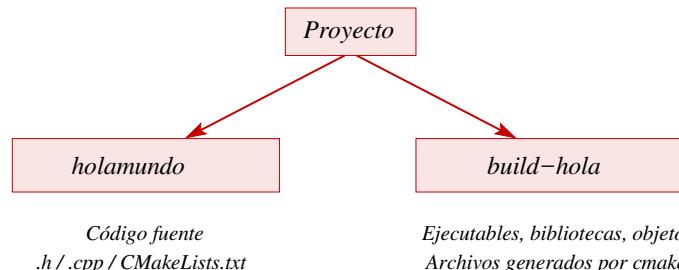


Figura A.1
Separar fuentes y binarios con *cmake*.

¹También podríamos hablar del IDE *CLion* multiplataforma para C/C++ —aunque no lo consideramos al no ser libre— que trabaja con *CMake*.

²Si le interesa, en la red puede encontrar distintos tutoriales y manuales para revisarlo en profundidad. Para empezar, también puede consultar directamente el manual *cmake-language* que seguramente tiene instalado en su sistema.

Si ha trabajado con un proyecto de cierto tamaño y sobre todo si tenía distintos módulos diferenciados se habrá dado cuenta de que organizar los fuentes en distintos directorios lo hace mucho más manejable. Además, si evitamos mezclar los binarios con los fuentes —por ejemplo, separar los objeto *.o* de los *cpp*— tendremos una solución más simple. Es más, si ha usado algún sistema de control de versiones —por ejemplo, *git*— para poder mantener los fuentes en un repositorio se habrá dado cuenta de que es ideal disponer de un directorio padre del que cuelgan exclusivamente todos los subdirectorios y fuentes que componen el proyecto.

Lo recomendable, por tanto, es separar en directorios independientes los fuentes de los binarios. Es cierto que podríamos pensar en una solución en la que los binarios estuvieran en un subdirectorio del árbol de fuentes pero es poco recomendable. Como aparece en la figura A.1, supondremos que creamos un directorio para todo el proyecto —en la figura “*Proyecto*” del que cuelgan como “*hermanos*” los fuentes y binarios; en nuestro ejemplo hemos supuesto, respectivamente, los subdirectorios *holamundo* y *build-hola*.

En la práctica, incluso podrá tener otros subdirectorios con binarios. Por ejemplo, puede mantener dos: uno con la compilación de archivos para poder trazar el código (compilación para depuración, modo *debug*) y otro con los archivos optimizados (compilación final, modo *release*).

Un ejemplo simple: un solo archivo

Como ejemplo, mostramos un proyecto con un solo archivo: «*hola_mundo.cpp*». Por supuesto, en este caso resulta innecesaria la gestión con *CMake*, pero sirve para hacer explícita la gestión básica de fuentes y binarios.

En primero lugar, creamos un directorio con nombre «*proyecto*» de donde colgará todo; dentro de éste, el directorio «*holamundo*» con los fuentes. Los fuentes serán el conocido archivo «*cpp*» que escribe el mensaje y un archivo *script* para «*cmake*» con nombre *CMakeLists.txt*. Por tanto, podemos hacer:



```
prompt: ~/proyecto> ls holamundo/
CMakeLists.txt hola_mundo.cpp
```

donde se pueden distinguir todos los archivos y directorios fuentes que componen nuestro proyecto. El contenido del archivo «*CMakeLists.txt*» para nuestro proyecto puede ser el siguiente:

```
1 # Archivo CMakeLists.txt básico para un programa de un fichero
2
3 PROJECT ( Hola )
4 CMAKE_MINIMUM_REQUIRED (VERSION 3.0)
5
6 ADD_EXECUTABLE ( hola_mundo hola_mundo.cpp )
```

donde se han usado sólo tres órdenes del lenguaje: «*PROJECT*», «*CMAKE_MINIMUM_REQUIRED*» y «*ADD_EXECUTABLE*». Todas en mayúscula, lo que nos permite distinguirlas claramente de los datos; aunque para el caso de los nombres de órdenes, el lenguaje no distingue minúsculas de mayúsculas. Precisamente por ello, algunos programadores prefieren las versiones en minúscula para poder diferenciarlas de los identificadores de variables.

Observe que no se ha incluido ningún detalle sobre qué compilador usar, dónde está ese compilador, las opciones de compilación o enlazado, etc. Sólo hemos especificado:

- Un comentario. Podemos usar el carácter «#» para añadir un comentario hasta final de línea.
- El proyecto se va a llamar «*Hola*». Cualquiera de nuestros proyectos va a tener una orden «*PROJECT*» en el archivo *CMakeLists.txt* del directorio raíz³.
- Debemos indicar qué versión de «*cmake*» se necesita. En este caso hemos puesto la versión 3.0, aunque por su simplicidad seguramente funcionará en cualquiera, incluso anteriores. Tenga en cuenta que esto es conveniente cuando usemos una característica que no exista en versiones anteriores. Con ello, podremos avisar al usuario de que actualice su programa «*cmake*» para el correspondiente proyecto.
- Hemos indicado que hay un ejecutable con nombre «*hola_mundo*» que se obtiene a partir del archivo «*hola_mundo.cpp*».

Con este simple archivo ya podemos resolver la compilación y generación del ejecutable correspondiente. Para ello, creamos un directorio «*build-hola*» independiente de los fuentes. Dentro de este directorio será donde ejecutemos «*cmake*». En concreto, podemos hacer:

³Realmente no tiene que ser así, incluso un proyecto puede estar compuesto por varios proyectos y por tanto contener varias veces esta orden, pero no será nuestro caso.

```

prompt:~/proyecto> mkdir build-hola
prompt:~/proyecto> cd build-hola
prompt:~/proyecto/build-hola> cmake ..../holamundo
-- The C compiler identification is GNU 5.4.0
-- The CXX compiler identification is GNU 5.4.0
... Otras líneas de comprobación ...
-- Configuring done
-- Generating done
-- Build files have been written to: /home/antonio/proyecto/build-hola

```

donde puede observar que para llamar a «`cmake`» basta con indicarle cuál es el directorio donde encontrar los fuentes. Cuando se lanza, interpreta las órdenes del archivo «`CMakeLists.txt`» y genera el resultado en una estructura de directorios que cuelga del directorio actual.

Por defecto va a generar los ficheros de configuración asociados a «`make`», es decir, archivos «`makefile`» de *UNIX* listos para generar los objetivos asociados a nuestro proyecto. Podemos escribir:

```

prompt:~/proyecto/build-hola> make
Scanning dependencies of target hola_mundo
[ 50%] Building CXX object CMakeFiles/hola_mundo.dir/hola_mundo.cpp.o
[100%] Linking CXX executable hola_mundo
[100%] Built target hola_mundo
prompt:~/proyecto/build-hola>

```

que obtiene el archivo ejecutable en el mismo directorio donde estamos, junto con otros archivos generados. Entre ellos, el archivo «`makefile`» que nos ha permitido la compilación. Este archivo tiene otros posibles objetivos configurados que puede listar así:

```

prompt:~/proyecto/build-hola> make help
The following are some of the valid targets for this Makefile:
... all (the default if no target is provided)
... clean
... depend
... edit_cache
... rebuild_cache
... hola_mundo
... hola_mundo.o
... hola_mundo.i
... hola_mundo.s
prompt:~/proyecto/build-hola>

```

A.2.2 Un proyecto con varios archivos fuente

En esta sección abordamos el problema de generar varios archivos ejecutables que se generan desde varios archivos fuente. Para no dedicar mucho tiempo a presentar el problema a resolver, usamos el mismo ejemplo que se presentó en el capítulo 2 (página 9)⁴. Los archivos son:

```

prompt:~/proyecto> ls geom/
bounding_box.cpp  circulo.h          CMakeLists.txt  punto.cpp  rectangulo.cpp
circulo.cpp       circulo_medio.cpp  longitud.cpp   punto.h    rectangulo.h

```

Este proyecto está compuesto de múltiples archivos y funciones que, si bien eran extremadamente simples, configuraban un buen ejemplo para explicar la compilación separada. Sin entrar en detalles de lo que contienen, es fácil intuir la relación que se creó con el siguiente archivo «`CMakeLists.txt`»:

⁵

⁴Si no dispone de los archivos, tampoco son necesarios para seguir la explicación, aunque no podrá probar los ejemplos.

```

1 # Archivo CMakeLists.txt para programas de geometría
2 PROJECT( GEOM )
3 cmake_minimum_required(VERSION 3.0)
4
5 add_executable( circulo_medio circulo_medio.cpp
6                  punto.cpp circulo.cpp )
7 add_executable( longitud longitud.cpp
8                  punto.cpp )
9 add_executable( bounding_box bounding_box.cpp
10                 punto.cpp rectangulo.cpp )
11

```

Con este archivo preparamos un directorio para los binarios y generamos los correspondientes archivos «makefile» como sigue:

```

Consola

prompt:~/proyecto> mkdir build-geom
prompt:~/proyecto> cd build-geom/
prompt:~/proyecto/build-geom> cmake ../geom
      ... líneas de comprobación y generación ...
-- Configuring done
-- Build files have been written to: /home/antonio/proyecto/build-geom

```

Ya podemos ejecutar «make» para obtener los ejecutables:

```

Consola

prompt:~/proyecto/build-geom> make
[ 9%] Building CXX object CMakeFiles/circulo_medio.dir/circulo_medio.cpp.o
[ 18%] Building CXX object CMakeFiles/circulo_medio.dir/punto.cpp.o
[ 27%] Building CXX object CMakeFiles/circulo_medio.dir/circulo.cpp.o
[ 36%] Linking CXX executable circulo_medio
[ 36%] Built target circulo_medio
[ 45%] Building CXX object CMakeFiles/longitud.dir/longitud.cpp.o
[ 54%] Building CXX object CMakeFiles/longitud.dir/punto.cpp.o
[ 63%] Linking CXX executable longitud
[ 63%] Built target longitud
[ 72%] Building CXX object CMakeFiles/bounding_box.dir/bounding_box.cpp.o
[ 81%] Building CXX object CMakeFiles/bounding_box.dir/punto.cpp.o
[ 90%] Building CXX object CMakeFiles/bounding_box.dir/rectangulo.cpp.o
[100%] Linking CXX executable bounding_box
[100%] Built target bounding_box

```

Observe los pasos que se han necesitado para la generación de los ejecutables: cada uno se ha resuelto de forma independiente lo que ha provocado la compilación repetida de algunos módulos.

Si especificamos cada fuente en cada objetivo, el sistema de «cmake» asume que debe compilarse para cada uno de ellos. No es un problema de «cmake», sino de que deberíamos haber reflejado que esos módulos corresponden a un recurso común para los ejecutables. Lo resolveremos en la sección A.2.4 (página 96).

Lo que sí es relevante en este punto es darse cuenta de cómo el generador ha establecido correctamente los pasos de compilación para obtener archivos objeto y los correspondientes enlazados para los ejecutables. Además, conoce qué archivos han dado lugar a qué objetivos, es decir, si modificamos un archivo fuente y ejecutamos «make» volverá a regenerar los objetivos afectados.

Ejercicio A.1 Pruebe las dos órdenes siguientes:

```

Consola

prompt:~/proyecto/build-geom> touch ../geom/circulo.cpp
prompt:~/proyecto/build-geom> make

```

para comprobar la regeneración que implica una modificación en el archivo «circulo.cpp». Después pruebe lo mismo con el archivo «punto.h» que afecta a todo el proyecto.

Añadimos una variable

El archivo «CMakeLists.txt» contiene una serie de archivos repetidos. Invita a escribirlos una única vez. Podemos hacerlo usando una variable. El formato de asignación de una variable normal es:

```
set( <variable> <valor>... )
```

donde hemos puesto puntos suspensivos porque podría haber una lista de valores. En nuestro ejemplo, usamos una variable para almacenar la lista de archivos fuente como sigue⁵:

```

1 PROJECT( GEOM )
2
3 cmake_minimum_required(VERSION 3.3)
4
5 set( GEOM_FUENTES punto.cpp circulo.cpp rectangulo.cpp )
6
7 add_executable( circulo_medio circulo_medio.cpp ${GEOM_FUENTES} )
8 add_executable( longitud longitud.cpp ${GEOM_FUENTES} )
9 add_executable( bounding_box bounding_box.cpp ${GEOM_FUENTES} )

```

En la línea 5 hemos incluido una orden para asignar un valor a la variable «GEOM_FUENTES», concretamente, una lista de 3 nombres de archivos fuente. En las tres últimas líneas indicamos que la orden contiene como último parámetro esa lista de archivos. Lógicamente, no es una buena solución, pues en este caso cada uno de los ejecutables tiene todos y cada uno de los archivos objeto, incluso los que no necesita. Tenga cuidado, no se deje llevar por la facilidad de «cmake» a la hora de crear objetivos.

Ejercicio A.2 Añada un nuevo ejecutable al archivo con nombre «longitud_simple» y que dependa únicamente de los dos archivos fuente «longitud.cpp» y «punto.cpp». Compare el tamaño de este ejecutable con el de «longitud».

Este ejemplo no sólo nos permite mostrar que debemos ser precisos al indicar los fuentes; además, muestra cómo podemos crear una variable en cualquier sitio simplemente dando un valor con la orden «set». Note la sintaxis para referirse al contenido de la variable usando el carácter «\$».

Debe tener cuidado con los nombres, pues en este caso sí se distinguen las minúsculas y mayúsculas. Por ejemplo, si en la última línea cambia una letra, el intérprete no se quejará, pues supondrá que es otra variable y asumirá que está vacía, es decir, no tiene nada. La consecuencia es que pensará que el ejecutable «bonding_box» se genera a partir de un único archivo fuente «bonding_box.cpp». El error aparecerá cuando intente compilar. Tal vez éste sea un buen motivo para que las variables se escriban completamente en mayúscula, primero porque “aparentan ser macros globales” y segundo porque la mezcla con minúsculas sería propensa a errores.

A.2.3 Variables, la caché y otras opciones

Una vez que ha entendido en qué consiste el desarrollo con *CMake*, es un buen momento para aclarar en qué consisten las variables, lo que es la caché y algunas opciones especialmente útiles incluso en sus proyectos más simples.

El intérprete que ejecuta las órdenes de «CMakeLists.txt» no sólo considera la secuencia de órdenes que aparecen, sino que maneja datos en forma de variables. Estas variables contendrán una cadena o lista de cadenas, como hemos visto en el ejemplo anterior. En un proyecto no sólo existen las variables que creamos nosotros, sino que también aparecerán múltiples valores que el intérprete necesita para generar el resultado. Por ejemplo, nada más incluir la orden «project» en nuestro archivo ya se han generado variables asociadas⁶. Podemos mostrar la información con la orden «message» como sigue:

```

1 PROJECT( GEOM )
2 cmake_minimum_required(VERSION 3.3)
3
4 message( "Valor de PROJECT_NAME: ${PROJECT_NAME}" )
5 message( "Valor de GEOM_SOURCE_DIR: ${GEOM_SOURCE_DIR}" )
6 message( "Valor de GEOM_BINARY_DIR: ${GEOM_BINARY_DIR}" )

```

que en nuestro proyecto ejemplo muestra lo siguiente:

```
Consola
Valor de PROJECT_NAME: GEOM
Valor de GEOM_SOURCE_DIR: /home/antonio/proyecto/geom
Valor de GEOM_BINARY_DIR: /home/antonio/proyecto/build-geom
```

No sólo vamos a tener las variables asociadas a nuestro proyecto, sino múltiples valores asociados al sistema con el que estamos trabajando:

- Variables para controlar los nombres de los programas que usaremos para compilar.
- Variables con los caminos —paths— donde encontrar archivos.
- Variables con la cadena concreta que indica las opciones de compilación.
- Etc.

⁵Note que aquí optamos por las órdenes en minúscula para diferenciar claramente las variables.

⁶Podríamos haber generado incluso más si hubiéramos asignado una versión al proyecto.

Puede consultar en el manual de *cmake-variables* que la lista de posibles variables en nuestro proyecto es muy amplia. A estas variables habrá que añadir cualquier otra que podamos crear, como el caso de «GEOM_FUENTES» que hemos visto más arriba. En principio, basta con algunos ejemplos para tener una idea intuitiva de cómo funcionan; puede trabajar con *CMake* incluso sin conocer mucho más, pues es fácil extraer el código que necesitamos desde ejemplos previos; en la sección A.3.1 volveremos sobre ellas para añadir algunos detalles que serán necesarios si tiene errores o el proyecto es más complejo.

La caché

En los ejemplos anteriores, hemos ejecutado «`cmake`» para generar una serie de archivos que nos permiten compilar nuestro proyecto con la orden «`make`». Para ello, se ha generado un archivo «`makefile`», pero no ha sido el único. Junto a él, también ha aparecido un archivo especialmente importante: «`CMakeCache.txt`».

En la sección anterior hemos usado una variable a la que hemos asignado un valor con la orden «`set`». Esta variable funciona como es habitual en un programa: en principio no tiene valor, se le asigna un primer valor, se usa y cuando acaba el programa, desaparece. Con la caché se establece un mecanismo para que el valor de las variables persista de una ejecución a otra.

Cuando se lanza «`cmake`» por primera vez, se crea el archivo «`CMakeCache.txt`» en el que se incluyen una serie de variables con el correspondiente valor. Esta caché se crea principalmente para:

- Mejorar el rendimiento. Cuando se lanza «`cmake`» por primera vez se deben realizar algunas operaciones de comprobación y localización en el sistema. Éstas pueden ser costosas por lo que repetirlas para cada ejecución resulta muy inefficiente. Si fijamos el resultado en la caché, las siguientes ejecuciones pueden usarla sin volver a calcularlas.
- Configurar parámetros del proyecto. La caché contiene valores que no vuelven a calcularse en cada ejecución. Si editamos la caché y cambiamos alguno de estos valores, las siguientes generaciones asumirán estos nuevos valores. Por tanto, resulta un sistema muy simple para cambiar algún parámetro sin tocar los archivos «`CMakeLists.txt`» originales.

Por ejemplo, puede descargar un paquete fuente, lanzar por primera vez «`cmake`» en un directorio de binario y editar la caché para ajustar los parámetros del proyecto para su sistema o sus preferencias particulares. Por ejemplo, indicando parámetros de compilación, directorios donde localizar o donde llevar resultados, módulos que quiere o no que se añadan al resultado final, etc.

Normalmente es muy sencilla la modificación, pues el archivo «`CMakeCache.txt`» es de texto y puede cambiarlo directamente en un editor. También puede hacerlo con alguna herramienta gráfica. Por ejemplo, si en mi máquina ejecuto el objetivo «`edit_cache`» de «`make`» se lanza la herramienta «`cmake-gui`» que ofrece una interfaz gráfica para la edición. Lo más recomendable es editarla como texto.

Podemos usar la orden «`set`» para asignar un valor a una variable que irá a la caché con un formato concreto:

```
set( <variable> <valor>... CACHE <tipo> <documentación> [FORCE] )
```

que corresponde a:

- La palabra «`CACHE`» para indicar que es una variable que se almacena en la caché.
- El tipo de dato. Puede indicar distintos valores: «`BOOL`», «`FILEPATH`», «`PATH`», «`STRING`» e «`INTERNAL`». Disponer del tipo facilita que «`cmake-gui`» ofrezca un diálogo adaptado al tipo de dato.
- Una cadena de documentación para el parámetro. Permite reconocer la intención de éste cuando editamos la caché.
- Opcionalmente podemos añadir la palabra «`FORCE`» para indicar que este parámetro se guardará en la caché pero se fijará en cada ejecución.

Por ejemplo, la variable «`CMAKE_BUILD_TYPE`» se añade a la caché cuando lanzamos por primera vez «`cmake`»; aparece con un valor vacío. Nos permite fijar distintos valores —lea la documentación en el mismo archivo «`CMakeCache.txt`»— entre los que se encuentran «`Debug`» o «`Release`». Si quiere, puede fijar el valor de esta variable en su archivo «`CMakeLists.txt`» con la siguiente línea:

```
1 PROJECT( GEOM )
2 cmake_minimum_required(VERSION 3.3)
3
4 set( CMAKE_BUILD_TYPE "Release" CACHE STRING "Fijamos Release" FORCE)
```

donde puede ver que no sólo hemos indicado que la generación obtendrá la versión final del software —opción «`Release`»— sino que indicamos «`FORCE`» de manera que se obliga ese valor; la edición de la caché no lo cambiará.

Algunas opciones que debería conocer

La primera opción que puede ser útil está relacionada con la caché: es la opción «`-D`» que podemos pasar a «`cmake`». Con esta opción podemos darle un valor a una variable de la caché. La sintaxis consiste en el nombre, el signo igual y el valor que le damos. Un ejemplo habitual es precisamente la que hemos visto en la sección anterior; con la siguiente línea podemos decir que genere los archivos para poder depurar:



de manera que no es necesario añadir nada al archivo «**CMakeLists.txt**». Note que si en éste hemos incluido la línea que vimos con la opción «**FORCE**» no servirá de nada la opción «**-D**». Realmente, en la práctica es mejor no forzar esta variable porque en un proyecto en desarrollo es probable que se quiera cambiar su valor en algún momento. Si se quiere uno u otro valor, siempre podemos usar la opción «**-D**»; tal vez, en una versión definitiva para lanzar se podría considerar forzar una de las opciones.

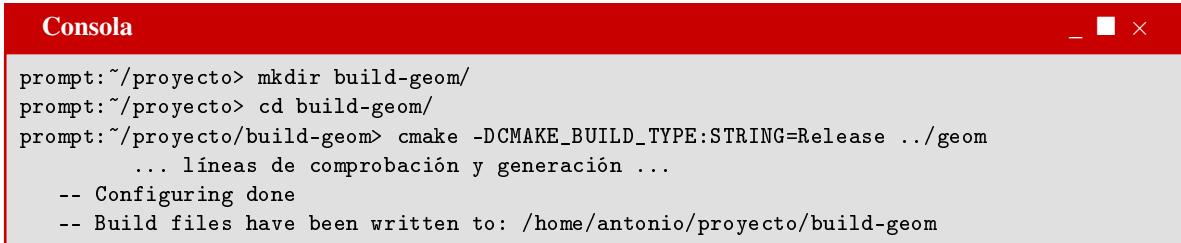
Aunque esta sintaxis es suficiente, también es posible indicar el tipo de la variable, después del nombre:



```
cmake -DCMAKE_BUILD_TYPE:STRING=Debug .. /geom
```

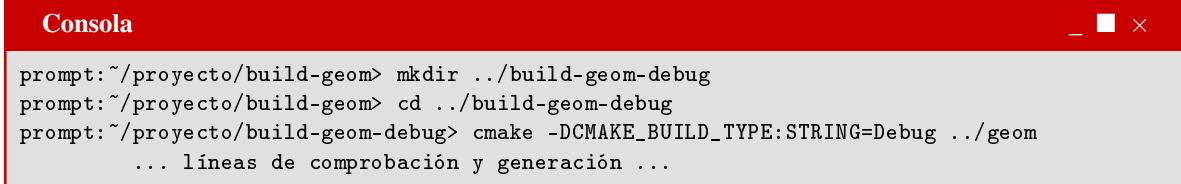
que puede ser útil en caso de que queramos editar la caché con algún programa o cuando sean nombres de archivos o directorios, en cuyo caso se crean los caminos absolutos.

El hecho de poder hacerlo en línea, de manera directa le puede permitir, por ejemplo, crear dos directorios de binarios. Podría hacer lo siguiente:



```
prompt:~/proyecto> mkdir build-geom/
prompt:~/proyecto> cd build-geom/
prompt:~/proyecto/build-geom> cmake -DCMAKE_BUILD_TYPE:STRING=Release .. /geom
... líneas de comprobación y generación ...
-- Configuring done
-- Build files have been written to: /home/antonio/proyecto/build-geom
```

para crear un directorio con los programas optimizados para ejecutar y también hacer:



```
prompt:~/proyecto/build-geom> mkdir .. /build-geom-debug
prompt:~/proyecto/build-geom> cd .. /build-geom-debug
prompt:~/proyecto/build-geom-debug> cmake -DCMAKE_BUILD_TYPE:STRING=Debug .. /geom
... líneas de comprobación y generación ...
```

lo que le permite trabajar con dos árboles de binarios, uno para las ejecuciones finales y otro para poder depurar.

Otra opción también asociada con «**cmake**» es «**-G**» que permite seleccionar un generador concreto. Por defecto, hemos usado “*Unix Makefiles*”. Escriba la orden «**cmake**» con esa opción para obtener un listado de las opciones disponibles en su sistema.

Finalmente, es conveniente indicar que podemos añadir una opción a «**make**» para tener más información sobre lo que está generando. Si observa los mensajes que obtenemos cuando lanzamos la compilación, verá que simplemente informa sobre el punto que está resolviendo y el porcentaje del proceso que está resuelto. Aunque resulta atractivo e ideal cuando todo funciona, es un problema cuando hay un error de compilación que rompe el proceso.

La solución para tener más información es la opción «**VERBOSE=1**». Note que no es una opción de «**cmake**», sino de «**make**». Obtendrá una salida muy larga y con demasiados detalles, pero podrá consultar exactamente las órdenes que está lanzando y descubrir qué detalle es el que puede estar impidiendo que se compile con éxito.

Realmente también se puede activar asignando la variable de entorno es «**VERBOSE**» o con una variable booleana que tiene en la caché; sin embargo, la opción de «**make**» es la más práctica si quiere hacer una ejecución puntual de esta forma.

A.2.4 Un proyecto con varios directorios y bibliotecas

Como ejemplo final de este pequeño tutorial sobre «**cmake**» completamos el proyecto «**GEO**M» organizando los archivos en directorios y configurando una biblioteca que contendrá los módulos reusables.

En primer lugar, proponemos una solución para organizar los ficheros fuente a distintos directorios. En concreto, creamos dos directorios: «**formas**» para incluir los tres módulos de manejo de formas y «**apps**» para incluir los programas que los aprovechan. El resultado es el siguiente:

```

prompt: ~/proyecto/build-geom> tree
|-- CMakeLists.txt
|-- apps
|   |-- bounding_box.cpp
|   |-- circulo_medio.cpp
|   '-- longitud.cpp
`-- formas
    |-- circulo.cpp
    |-- circulo.h
    |-- punto.cpp
    |-- punto.h
    '-- rectangulo.cpp
        '-- rectangulo.h

```

Con esta estructura, podríamos usar «`cmake`» para generar un resultado similar al que hemos visto en las secciones anteriores. El archivo «`CMakeLists.txt`» puede ser el siguiente:

```

1 PROJECT ( GEOM )
2 CMAKE_MINIMUM_REQUIRED (VERSION 3.0)
3
4 include_directories ( ${GEOM_SOURCE_DIR}/formas )
5
6 aux_source_directory( formas FORMAS_LIST )
7
8 add_executable ( circulo_medio apps/circulo_medio.cpp ${FORMAS_LIST} )
9 add_executable ( longitud apps/longitud.cpp ${FORMAS_LIST} )
10 add_executable ( bounding_box apps/bounding_box.cpp ${FORMAS_LIST} )

```

En este listado hemos incluido dos órdenes nuevas:

- «`INCLUDE_DIRECTORIES`». Indicamos un directorio donde podrá encontrar archivos cabecera que se incluirán con `#include`. Es necesario porque ahora los archivos en el directorio «`apps`» no están con los archivo cabecera. Observe cómo hemos aprovechado la variable que nos dice dónde está el directorio fuente del proyecto.
- «`AUX_SOURCE_DIRECTORY`». Tiene dos parámetros: un directorio y una variable. Asigna a la variable los archivos fuente que hay en el directorio. Cada nombre incluye la ruta hasta él, en nuestro caso comienzo con «`formas/`».

Si crea un directorio para binarios y compila esta solución verá que obtenemos algo similar a lo que teníamos antes. De nuevo, los ejecutables incluyen todos los objetos.

Múltiples archivos `CMakeLists.txt`: una biblioteca

La solución ideal para nuestro proyecto es crear una biblioteca, como vimos en el capítulo 2 (página 9). Esta biblioteca se puede considerar un módulo independiente de nuestro proyecto. Para resolver el problema de la biblioteca, no necesitamos conocer qué programas la usarán, de igual forma que crear un programa no implica conocer los detalles de cómo se crea la biblioteca.

Creamos dos nuevos archivos «`CMakeLists.txt`» para dividir nuestro proyecto en dos partes diferenciadas. Ahora tenemos los archivos fuente en el mismo sitio, pero tres archivos `CMake`:

```

-- CMakeLists.txt
-- apps
|   '-- CMakeLists.txt
`-- formas
    '-- CMakeLists.txt

```

El nuevo archivo «`CMakeLists.txt`» para el proyecto contiene los parámetros globales y la llamada a procesar los directorios. El código es:

```

1 PROJECT ( GEOM )
2 CMAKE_MINIMUM_REQUIRED (VERSION 3.0)
3
4 include_directories ( ${GEOM_SOURCE_DIR}/formas )
5
6 set ( LIBRARY_OUTPUT_PATH ${GEOM_BINARY_DIR}/lib CACHE PATH
7           "Output directory for the GEOM libraries" )
8
9 link_directories ( ${LIBRARY_OUTPUT_PATH} )
10

```

```

11 # En formas hay una librería que construir
12 add_subdirectory (formas)
13
14 # En apps hay ejecutables que crear
15 add_subdirectory (apps)

```

Note que hemos creado una variable «LIBRARY_OUTPUT_PATH» que aparecerá en la caché y que contiene el directorio donde se añadirán las bibliotecas que se creen. Además, hemos añadido dos órdenes nuevas:

- «link_directories». Indica un directorio donde encontrará bibliotecas con las que enlazar. Cuando tenga que crear un ejecutable, añadirá este directorio en las rutas de búsqueda.
- «add_subdirectory». Indica que debe procesar un nuevo archivo «CMakeLists.txt» situado en ese directorio.

Realmente, los subdirectorios podrían contener un archivo «CMakeLists.txt» completo, incluyendo una orden «PROJECT» para configurarlo como un subproyecto dentro del actual. Esto podría reflejarse como un proyecto independiente si genera la configuración para un *IDE*.

Sin embargo, vamos a crear un nuevo archivo sin cambiar de proyecto. El archivo que tenemos en el directorio «formas» se ocupa solamente de crear la biblioteca a partir de los fuentes que hay en el directorio. El archivo completo es:

```

1 # CMakeLists.txt para crear la biblioteca del proyecto
2 set ( FORMAS_SRCS
3         punto.h     punto.cpp
4         circulo.h    circulo.cpp
5         rectangulo.h rectangulo.cpp
6     )
7
8 # Creamos la biblioteca formas
9 add_library (formas ${FORMAS_SRCS})

```

Hemos usado una nueva orden:

- «add_library». Recibe el nombre de una nueva biblioteca junto con la lista de fuentes que la componen. En consecuencia generará el correspondiente archivo binario en el directorio que hemos configurado en el padre.

Es interesante que note los archivos que se han listado en los fuentes de la biblioteca. En este caso no sólo hemos incluido los archivos «.cpp», sino que también hemos incluido los «.h». No es algo especial de las bibliotecas. Cuando creamos una lista de fuentes podemos reunir los dos tipos de archivos en una misma lista. Se pueden compilar todos los archivos, «cmake» sólo generará los objetivos para los archivos «.cpp».

Esta inclusión de los archivos cabecera en la lista de fuentes podríamos haberla realizado también en los ejemplos anteriores. Es cierto que incluir los archivos «.cpp» es suficiente —de hecho, aquí también basta con ellos— pero es conveniente incluirlos todos. De esta forma es más sencillo procesar que la lista de fuentes que componen la biblioteca incluye también esos archivos, aunque no haya que compilarlos. Por tanto, en adelante, inclúyelos en todos los casos.

Finalmente, nos falta el archivo «CMakeLists.txt» del directorio «apps» y que configura los objetivos para los programas. El listado es el siguiente:

```

1 # Aplicaciones del proyecto GEOM
2
3 add_executable ( circulo_medio circulo_medio.cpp )
4 target_link_libraries ( circulo_medio formas )
5
6 add_executable ( longitud longitud.cpp )
7 target_link_libraries ( longitud formas )
8
9 add_executable ( bounding_box bounding_box.cpp )
10 target_link_libraries ( bounding_box formas )

```

Observe que, de nuevo, es un archivo que será parte del proyecto «GEOM» del directorio padre. En este caso sólo añadimos los ejecutables que tendrán que compilar el archivo «.cpp» que contiene la función **main**. En este caso, aparece una nueva orden:

- «target_link_libraries». Indica que la compilación de ese objetivo requiere en enlace con la biblioteca que se indica. Por tanto, los tres ejecutables se compilarán enlazando con la biblioteca «formas».

Además, esta orden no se usa únicamente para indicar las bibliotecas con las que enlazar un archivo para obtener un ejecutable sino que también puede expresar dependencias entre bibliotecas. Si la biblioteca «formas» dependiera de una segunda, se hubiera usado esta orden cuando se creó la primera de manera que estos ejecutables dependerían, indirectamente, de la segunda biblioteca.

Ejercicio A.3 Compruebe el funcionamiento de la última solución que hemos dado para el proyecto «GEOM» en directorios y con una biblioteca. Compruebe las operaciones que se realizan si modifica alguno de los fuentes: alguna aplicación, algún archivo «.cpp» de la biblioteca o algún archivo de cabecera de ésta, por ejemplo, «punto.h».

Re-Ejecución de `cmake`

Por último, es importante aclarar la forma de trabajar con un proyecto gestionado con *CMake*, concretamente, cuándo hay que ejecutar la orden «`cmake`» y la orden «`make`». Para comenzar hay que crear un directorio de binarios y lanzar la orden «`cmake`» para establecer el estado inicial. Después de esto, podemos lanzar cualquiera de las dos.

En principio, lo único que necesitamos es lanzar la orden «`make`» cada vez que se realice una modificación. Si cambian los fuentes, es necesario volver a compilar; pero si cambian los archivos *CMakeLists.txt* habría que volver a regenerar los archivos de construcción del proyecto. Esta regeneración se realiza automáticamente cuando lanzamos «`make`».

A pesar de ello, es posible encontrarse con algún caso en que tenga que lanzar de nuevo «`cmake`». Por ejemplo, consulte la orden «`aux_source_directory`» del manual. En estos casos, no debe temer que se pierdan datos: los datos que hay en la caché y que hemos modificado siguen con el mismo valor y los binarios compilados que no necesitan recompilarse siguen disponibles.

A.3 El lenguaje de CMake

Si quiere usar *CMake*, el tutorial de las secciones anteriores puede ser una buena introducción que le permita crear sus primeras soluciones sin necesidad de entrar en más detalles. Sin embargo, es probable que desee conocer más, que tenga problemas si su proyecto se ha complicado o que quiera leer código de terceros.

La intención de esta sección es incluir alguna información adicional que aclare detalles de cómo funciona realmente el lenguaje, aunque sin intención de recorrer todo su potencial. Tenga en cuenta que una vez que conozca lo más básico, puede usar el manual que se ofrece junto con el programa para consultar detalles.

A.3.1 Variables

Son el elemento fundamental para almacenar cualquier dato aunque están simplificadas hasta el punto de que todas son cadenas. El nombre de una variable es sensible a mayúsculas y minúsculas; si consulta ejemplos de archivos *CMakeLists*, verá que normalmente las variables se escriben con todas sus letras en mayúscula.

Cualquier variable tiene una cadena, no hay nada que declarar. Si se quiere usar el valor de una variable que nunca ha aparecido simplemente está vacía. Ésta es la normal general, pero para poder manejar distintas posibilidades hay que entender cómo funcionan estas cadenas. En esta sección vamos a presentar las líneas generales más relevantes.

Asignación: cadenas

Para asignar un valor a una variable se usa la orden «`set`». Como hemos visto, tiene un primer parámetro que corresponde al nombre de la variable y un segundo que es la cadena a asignar. Si queremos que deje de tener ese valor podemos usar «`UNSET`».

¿Cómo se escribe una cadena? El formato de escritura de cadenas tiene distintas alternativas. En principio, la forma más sencilla es ponerlo entre comillas dobles, aunque éstas no son obligatorias; se entiende que si se espera una cadena, lo que aparece lo debe interpretar de esa forma:

```
1 set ( msg "Hola" )
2 set ( msg Hola ) # Obtenemos el mismo resultado
```

anque si lo que quiere es incluir espacios, tendrá que optar por incluir las comillas.

El acceso al contenido de una variable se hace con la sintaxis « `${nombre}`». Por ejemplo:

```
1 set ( msg "Hola mundo" )
2 message ( ${msg} )
```

Teniendo en cuenta que las variables no se declaran, si usa un nombre que no existe no provoca ningún error, se considera la cadena vacía. Deberá tener cuidado con las mayúsculas, pues si el nombre no coincide exactamente el valor que referencia probablemente está vacío y a simple vista no es sencillo verlo si mezcla minúsculas y mayúsculas. Por ejemplo, las siguientes líneas:

```
1 set ( msg "Hola mundo" )
2 message ( ${Msg} )
```

provocan un error puesto que el intérprete sustituye el valor de «`Msg`» que es vacío lo que implica una orden «`message`» sin parámetros.

Tal vez resulta llamativo el error, pues sí que tiene un parámetro: la cadena vacía. No es así, cuando el intérprete encuentra una referencia de este tipo, la sustituye por su contenido lo que provoca la re-interpretación del resultado tras la sustitución. Por ejemplo, si escribe:

```
1 set ( MSG "Hola mundo" )
2 set ( m M )
3 set ( s SG )
4 message ( "El mensaje en \${MSG} es \${\${m}\${s}}" )
```

provocará la siguiente salida:

```
El mensaje en ${MSG} es Hola mundo
```

Observe que hemos tenido que “*escapar*” el carácter «\$» para que no lo interprete como un acceso al contenido de la variable. Por supuesto, podrá encontrar otros caracteres que necesitan de esa barra para que sean correctamente interpretados. Incluso puede escapar el espacio:

```
1 set ( MSG Hola\ mundo )
```

de forma que ahora no es necesario añadir las comillas. ¿Y si no lo “escapo”? Lea la siguiente sección.

Listas

Es habitual tener casos de listas, por ejemplo, una lista de archivos o de opciones. Una variable puede contener una cadena, no es un vector de cadenas. Esto se complica, pero «cmake» lo resuelve sin saltarse la norma de que una variable es una cadena.

La solución es sencilla, podemos almacenar una lista de cadenas si determinamos un carácter delimitador: el carácter «;» hace ese papel. La sintaxis debe tender a ser simple y legible; el intérprete va a pasar de lista a cadena y de cadena a lista de forma casi transparente; a pesar de eso, tendremos que ser conscientes, pues si hay un error es importante conocer los detalles. Por ejemplo, si escribe:

```
1 set ( SRCS f1.cpp f2.cpp )
2 set ( SRCS ${SRCS} fuente3.cpp )
3 message ( ${SRCS} )
```

realmente no se han realizado asignaciones de múltiples valores en «SRCS», sino que se ha interpretado una lista de valores. Puede pensar que las dos primeras líneas son:

```
1 set ( SRCS f1.cpp;f2.cpp )
2 set ( SRCS f1.cpp;f2.cpp;fuente3.cpp )
```

porque si el parámetro consta de una cadena y ha puesto varias cosas seguidas, es una cadena con todas ellas encadenadas con el carácter «;» para delimitar los distintos ítems de la lista. El resultado en la salida es:

```
f1.cppf2.cppf3.cpp
```

porque escribir una cadena que codifica una lista es escribir todos sus componentes. Si quiere escribir el carácter tendrá que usar «\;» para que el intérprete no lo considere el separador:

```
1 set ( fuentes "f1.cpp;f2.cpp" ) # Igual sin comillas
2 message ( ${fuentes} )
3 set ( fuentes "f1.cpp\f2.cpp" )
4 message ( ${fuentes} )
```

lo que escribe:

```
f1.cppf2.cpp
f1.cpp;f2.cpp
```

consecuencia de que en el primer caso imprime una lista de dos elementos y en el segundo una lista con un único elemento que contiene un carácter literal ‘;’.

En cualquier caso, si bien estos ejemplos sirven para “jugar” con esa interpretación del carácter como separador, en la mayoría de los casos podrá olvidar que existe. Cuando quiera imprimir el contenido de una lista, simplemente escríbalos entre comillas. El código que usará en sus archivos tendrá el siguiente aspecto:

```
1 set ( fuentes f1.cpp f2.cpp )      # Creamos una lista sin pensar en ;
2 message ( "${fuentes}" )          # Imprimimos la lista con ;
3 list ( LENGTH fuentes NUMITEMS ) # El nombre de la lista no incluye $
4 message ( "La lista ${fuentes} tiene ${NUMITEMS} elementos" )
```

que provoca una salida como la siguiente:

```
f1.cpp;f2.cpp
La lista f1.cpp;f2.cpp tiene 2 elementos
```

donde no debe sorprenderse por la presencia del carácter separador de los elementos de la lista. Finalmente, es interesante indicar que el lenguaje permite realizar otras operaciones con la lista. Si lo desea, consulte la orden «list» en el manual.

Variables numéricas

También podemos manejar variables numéricas. De nuevo, una variable contiene una cadena, pero el intérprete puede extraer el número si lo usamos en ese contexto. Por ejemplo, podemos hacer:

```
1 set ( numero 10 )
2 math ( EXPR res "${numero} + 1" )
3 message ( ${res} )
```

que escribe el número 11, aunque realmente es una cadena guardada en la variable «res».

Si quiere reflexionar un poco más sobre esta sintaxis, puede probar las siguientes órdenes:

```
1 set ( numero 10 )
2 set ( EXPRESION EXPR res "${numero} + 1" )
3 message ( "${EXPRESION}" )
4 math ( ${EXPRESION} )
5 message ( ${res} )
```

que obtienen el siguiente resultado

```
EXPR;res;10 + 1
11
```

Observe que «math» no es más que una orden que espera tres parámetros; además, hemos tenido que poner entre comillas la expresión para que forme una única cadena, porque tiene espacios. Si elimina los espacios, también puede eliminar las comillas. Si elimina las comillas pero no los espacios, obtiene:

```
EXPR;res;10;+;1
CMake Error at CMakeLists.txt:16 (MATH):
  MATH EXPR called with incorrect arguments.
```

Variables booleanas

Si se interpreta la cadena que contiene una variable como un valor booleano, se considera que las cadena vacía o con valores «0, FALSE, OFF, NO, IGNORE» y cualquier cadena terminada en «-NOTFOUND» son valores «false», incluso si están en minúsculas.

Variables de entorno

El caso de las variables de entorno es algo especial, porque es el único caso que se considera una estructura tipo *array asociativo*. Con la sintaxis «\$ENV{nombre}» puede asignar o consultar el valor de una variable de entorno.

El uso es como el resto de variables, aunque aquí resultará un poco extraño ver que el nombre de la variable contiene las llaves. Por ejemplo, podría imprimir el contenido de la variable «\$ENV{HOME}» (note que accedemos al valor) o asignar un valor a la variable «ENV{PATH}» (note que no accedemos sino que sólo indicamos el nombre).

A.3.2 Ámbito de las variables

Sabemos cómo funcionan las variables pero ¿dónde se conocen? Es una pregunta habitual cuando se comienza con *CMake*, puesto que disponemos de distintos archivos *CMakeLists.txt* y cada uno de ellos crea sus propias variables.

En principio, la idea es que las variables sean globales a cada archivo *CMakeLists.txt*. Sin embargo, debemos tener en cuenta que el fuente está distribuido en distintos directorios, lo que hace que los archivos *CMakeLists.txt* se procesen de forma ordenada con una estructura jerárquica. En consecuencia, las variables tienen un ámbito que denominamos de directorio pero además cada vez que se procesa un archivo *CMakeLists.txt* se copian las variables del “directorío padre”.

Además, es posible añadir un parámetro —que es opcional en la orden «set»— para indicar que debería añadirse al ámbito del padre. Con esta opción, la sintaxis para una variable normal es la siguiente:

```
set ( <variable> <valor>... PARENT_SCOPE )
```

Por otro lado, no podemos olvidar que el lenguaje también dispone de “funciones”, es decir, podemos crear nuestras propias órdenes con «function». Éstas crean un nuevo ámbito para que las variables sean locales⁷

Es necesario indicar que cuando usamos el parámetro «PARENT_SCOPE», no nos referimos únicamente al directorio padre, sino también al ámbito padre de llamada a una función.

⁷Piense que la reutilización de estas funciones en distintos archivos —o incluso proyectos— podría fácilmente crear un problema de colisión de nombres.

La caché

La caché constituye un ámbito de variables especial distinto a los anteriores de directorio o de función. El objetivo de la caché es disponer de los valores calculados para la siguiente ejecución —lo que permite que el usuario pueda ajustar cualquier valor “a mano” editando un fichero de valores— por lo que podemos asignar un valor a una variable sin saber nada de cómo funciona *CMake*, pudiendo consultar el contenido de una variable cuando se obtiene en cualquier momento.

Podemos decir que su ámbito es global porque disponemos de ellas en cualquier punto, aunque tendremos que tener en cuenta que en caso de colisión no se podrán consultar. Lo aconsejable es que intente evitar que los nombres coincidan para que no tenga problemas. El orden de consulta es:

1. Variables en la función.
2. Variables de directorio.
3. Variables de caché.

sin olvidar que si no se encuentra en ningún sitio se considera una variable con valor cadena vacía.

Para situar una variable en la caché, podemos usar la orden «`set`» con una sintaxis adaptada como hemos visto anteriormente.

A.3.3 Condicionales

La orden «`if/elseif/else/endif`» se puede usar para ejecutar un grupo de órdenes de manera condicional. El formato sigue el siguiente esquema:

```
if ( <expresion1> )
...
elseif ( <expresion2> )
...
else ( <expresion1> )
...
endif( <expresion1> )
```

donde las expresiones corresponden a valores booleanos. Pueden ser variables booleanas (véase la sección anterior) o expresiones más complejas. Consulte el manual de «`if`» para ver múltiples posibilidades. Cuando use esta orden es interesante que recuerde dos detalles:

- En la parte «`else`» y «`endif`» se repite la misma expresión que corresponde a la orden «`if`». Nos permite asociar más fácilmente una líneas con otras. Si el intérprete encuentra una expresión distinta dará un aviso.
- Las variables que aparecen en la expresión no incluyen el carácter «`$`» para obtener su valor. Si el intérprete encuentra un nombre que no corresponde a un valor booleano de los que hemos visto en la sección anterior, entiende que es una variable.

Por ejemplo, podemos añadir a nuestros ficheros *CMake* una comprobación del número de archivos que se están añadiendo a los ejecutables para avisar sobre la creación de una biblioteca:

```
1 list ( LENGTH FORMAS_SRCS NUMITEMS )
2 if ( NUMITEMS GREATER 10 )
3   message ( WARNING "Se ha detectado un enlazado con ${NUMITEMS} archivos" )
4   message ( WARNING "    debería considerar la posibilidad de crear una
5     biblioteca" )
6 endif ( NUMITEMS GREATER 10 )
```

donde hemos indicado además el parámetro «`WARNING`» para que aparezca en la salida con un aviso y no como mera información.

Podemos anidar varias órdenes «`if`». Por ejemplo, para avisar de que podría añadir una opción:

```
1 if ( DEFINED CMAKE_BUILD_TYPE )
2   message( "Dispone de la variable para seleccionar el tipo" )
3   if ( NOT CMAKE_BUILD_TYPE )
4     message( WARNING "    ... pero está vacío" )
5   else ( NOT CMAKE_BUILD_TYPE )
6     message( "    ... y tiene el valor ${CMAKE_BUILD_TYPE}" )
7   endif ( NOT CMAKE_BUILD_TYPE )
8 endif( DEFINED CMAKE_BUILD_TYPE )
```

donde hemos aprovechado que la cadena vacía se considera un valor booleano de falso. Si quiere escribirlo de forma más explícita, podríamos haber preguntado por la igualdad como sigue:

```
1 # ...
2 if ( CMAKE_BUILD_TYPE STREQUAL "" )
3 # ...
```

También podemos preguntarnos por la existencia de un directorio para obtener ciertos datos que necesita el proyecto:

```

1 if (NOT DATA_GEOM_DIR) # Debemos disponer de este directorio para seguir
2 set (DATA_GEOM_DIR "$ENV{DATA_GEOM_DIR}" ) # Tal vez en el entorno
3 if ( "${DATA_GEOM_DIR}" STREQUAL "" )
4   message ( "Variable DATA_GEOM_DIR no asignada. Puedes:" )
5   message ( " Crear una variable de entorno:" )
6   message ( " export DATA_GEOM_DIR=<ruta_a_datos>" )
7   message ( " Usar la opción -D al hacer cmake:" )
8   message ( " cmake -DDATA_GEOM_DIR:PATH=<ruta_a_datos>" )
9   # Cortamos la generación para que el usuario corrija el problema
10  message ( FATAL_ERROR "No es posible seguir sin la asignación" )
11 endif ( "${DATA_GEOM_DIR}" STREQUAL "" )
12 endif (NOT DATA_GEOM_DIR)

```

También podemos asegurarnos de que el estándar de C++ que usamos es el 14, pero especificando unos *flags* muy concretos en el caso de ser el compilador de la *GNU*:

```

1 if ( CMAKE_COMPILER_IS_GNUCXX )
2   message ( "Especificamos flags concretos para la gnu..." )
3   set ( GEOM_EXTRA_CMAKE_CXX_FLAGS "-Wall -Wextra -pedantic -std=c++14"
4         CACHE STRING "Flags extra que se añaden a CMAKE_CXX_FLAGS" )
5   set ( CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${GEOM_EXTRA_CMAKE_CXX_FLAGS}" )
6 else() # Si estamos con otro, al menos fijamos que es C++14
7   set ( CMAKE_CXX_STANDARD 14 )
8 endif ( CMAKE_COMPILER_IS_GNUCXX )

```

Observe que en este ejemplo nos hemos ahorrado repetir la expresión en la parte «*else*».

A.3.4 Bucles

También disponemos de distintas alternativas para escribir bucles. La primera, un bucle «*foreach*» para iterar sobre una serie de elementos o como bucle contador. Tiene la forma:

```

foreach( <variable> ... )
...
endforeach( <variable> )

```

donde hemos dejado con puntos suspensivos la primera línea porque tenemos varias posibilidades: una lista, un rango de valores, un rango con saltos, etc. Consulte el manual de «*foreach*» si desea más detalles. Por ejemplo, podríamos usarlo para iterar sobre los archivos de una lista:

```

1 # ...
2 aux_source_directory(apps EXE_LIST)
3
4 foreach ( nombre ${EXE_LIST} )
5   get_filename_component(nombre_simple ${nombre} NAME_WE)
6   message ( "Configuramos ${nombre_simple} desde ${nombre}" )
7   add_executable( ${nombre_simple} ${nombre} ${FORMAS_LIST} )
8 endforeach()

```

donde hemos usado la orden «*get_filename_component*» para extraer el nombre simple que corresponde a un nombre de archivo con ruta y extensión.

Por otro lado, podemos crear un bucle controlado por una variable expresión booleana. La forma es la siguiente:

```

while( <condición> )
...
endwhile( <condición> )

```

y un ejemplo de su uso el siguiente:

```

1 set ( number 1 )
2 while ( number GREATER 0 AND number LESS 11 )
3   message( "Iteración con ${number}" )
4   math ( EXPR number "${number} + 1" ) # decrement number
5 endwhile ( number GREATER 0 AND number LESS 11 )

```

anque si queremos recorrer los elementos desde el 1 al 10 podemos usar el bucle «*foreach*»:

```

1 foreach ( number RANGE 1 10 )
2   message( "Iteración con ${number}" )
3 endforeach ( number )

```

A.3.5 Macros y funciones

Se pueden crear macros y funciones para facilitar la reutilización de código o mejorar el diseño de nuestras soluciones. Las dos opciones son muy similares y de hecho tienen una sintaxis prácticamente idéntica. Para las macros:

```
macro ( <nombre> [<parám1> [<parám2> [<parám3> ...]] )
...
endmacro ( <nombre> )
```

donde usamos las palabras «macro» y «endmacro», y para las funciones:

```
function ( <nombre> [<parám1> [<parám2> [<parám3> ...]] )
...
endfunction ( <nombre> )
```

En ambos casos, la llamada se hace simplemente escribiendo el nombre de la función o la macro y entre paréntesis la lista de argumentos.

Las macros se crearon antes en el lenguaje y tienen un funcionamiento más simplificado. Funcionan de forma similar a las macros de C/C++ en las que simplemente se hace una sustitución. Las funciones son más parecidas a las funciones de C/C++; recuerde que una llamada a la función crea un nuevo ámbito de las variables. En consecuencia, debería tener cuidado con las macros, puesto que los nombres de las variables podrían colisionar con otras.

Para mostrar la diferencia y a modo de ejemplo, puede considerar el siguiente trozo de código:

```
1 # Este código no se ejecuta hasta que llamemos a la macro
2 macro ( Macro_prueba param1 )
3   message ( "El parámetro que ha pasado a la macro es: ${param1}" )
4   set ( param1 "adiós" )
5   message ( "El parámetro cambiado en la macro: ${param1}" )
6 endmacro ( Macro_prueba )
7
8 # Este código no se ejecuta hasta que llamemos a la función
9 function ( Funcion_prueba param1 )
10  message ( "El parámetro que ha pasado a la función es: ${param1}" )
11  set ( param1 "adiós" )
12  message ( "El parámetro cambiado en la función: ${param1}" )
13 endfunction ( Funcion_prueba )
14
15 # Llamamos a la macro y a la función
16 Macro_prueba ( hola )
17 Funcion_prueba ( hola )
```

que obtiene los siguientes mensajes:

```
Consola
El parámetro que ha pasado a la macro es: hola
El parámetro cambiado en la macro: hola
El parámetro que ha pasado a la función es: hola
El parámetro cambiado en la función: adiós
```

Tanto para las macros como las funciones, es posible llamarlas con más parámetros de los declarados. En ese caso, puede usar las variables predefinidas «ARGC» que contiene el número de parámetros y las variables «ARGV0», «ARGV1», «ARGV2», etc. con los valores. Además, para facilitar su gestión con listas, «ARGV» es una lista con todos y «ARGN» es la lista con todos los “parámetros extra” pasados.

A.4 Módulos

El código fuente para gestionar un proyecto no está sólo en archivos con «**CMakeLists.txt**». Además, puede encontrar:

- Archivos *script* con extensión «**.cmake**» para que se ejecuten con la orden «**cmake -P**». El código de estos *scripts* no está diseñado para gestionar un proyecto, no es más que una serie de órdenes del lenguaje que se ejecutan cuando lo lanzamos. Algunos de los ejemplos anteriores pueden probarse de esta forma.
- Archivos que corresponden a un *módulo* también con extensión «**.cmake**». Están diseñados para incluirse con la orden «**include**».

En esta sección se muestra el uso habitual de módulos para poder relacionar distintos paquetes. Los programas normalmente necesitan de bibliotecas, programas u otros archivos externos para poderse compilar. Si queremos una solución independiente del sistema, se debería facilitar la inclusión de esos paquetes externos.

No sólo de terceros, sino para nuestros propios proyectos. Imagine que crea una biblioteca y quiere ofrecerla a otros grupos de trabajo. Lo mejor es facilitarles la labor de integrar la biblioteca creada en esos otros proyectos.

A.4.1 La orden `find_package`

La forma más sencilla de usar otros paquetes es con la orden «`find_package`». Esta orden recibe un parámetro «XXX» con el nombre del paquete y se ocupa de localizar un archivo con nombre «`findXXX.cmake`» para incluirlo en el proyecto.

Si ha instalado en su sistema los recursos para trabajar con *CMake* seguramente también se han instalado archivos adicionales entre los que se encuentran archivos «`findXXX.cmake`». Directamente, *CMake* ya ofrece cientos de archivos preparados para incluir paquetes externos. Los paquetes más ampliamente usados están cubiertos con estos fuentes. Además, puede encontrar otros en la red o incluso dentro de los kits de desarrollo de esos paquetes.

La misma orden «`cmake`» ofrece una opción «`--help-module`» para obtener una ayuda o puede buscar el archivo «`findXXX.cmake`» que probablemente contendrá comentarios sobre su uso.

Si usa esta orden para buscar un módulo «XXX» y lo encuentra, probablemente se definirán una serie de variables para que pueda usarlas en su proyecto. Normalmente tienen nombres predefinidos, entre los que podemos destacar:

- «`XXX_FOUND`». Nos permite saber si se ha encontrado el módulo.
- «`XXX_INCLUDE_DIRS`». Los directorios con archivos cabecera para la compilación
- «`XXX_DEFINITIONS`». Las definiciones que se deben incluir para compilar con el módulo.
- «`XXX_LIBRARIES`». Las bibliotecas con las que hay que enlazar nuestros programas.

La sintaxis de la orden «`find_package`» tiene muchas posibilidades, aunque desde un punto de vista práctico se suelen usar relativamente pocas. Tal vez —como indica el manual— la versión simplificada que puede tener en cuenta es:

```
find_package( <package> [major[.minor]] [EXACT] [REQUIRED|QUIET] )
```

donde probablemente la opción más interesante sea «`REQUIRED`» que indica que si no se encuentra el paquete debería dar un error y parar. Será necesario si nuestro proyecto no puede compilarse sin el paquete correspondiente.

Como ejemplo, suponga que desarrolla una aplicación que lee imágenes con formato *PNG*. En internet puede encontrar fácilmente las bibliotecas que permiten leer este tipo de formato. Son bibliotecas en C que se pueden usar en nuestros proyectos C++. Para usarlas, sólo tenemos que buscar con «`find_package`» como sigue:

```
1 find_package ( PNG )
2 if( PNG_FOUND )
3   include_directories( ${PNG_INCLUDE_DIRS} )
4   add_definitions( ${PNG_DEFINITIONS} )
5   # Otras líneas para enlazar con PNG... por ejemplo:
6   target_link_libraries( xxx ${PNG_LIBRARIES} ) # Para algún target xxx
7 endif( PNG_FOUND )
```

Observe que no hemos indicado nada «`REQUIRED`», lo que indica que no es un paquete necesario. Se supone que nuestro proyecto puede compilarse sin esta biblioteca. Probablemente el resultado ofrecerá menos posibilidades. Por ejemplo, nuestro proyecto puede ser un visor de imágenes que sin este paquete no podrá leer el formato *PNG* pero sí otros; imagine que sí que ha encontrado el paquete *JPEG*.

Finalmente, tenga en cuenta que puede crear sus propios módulos de búsqueda. Si crea un nuevo paquete y desea que otros lo usen, puede ofrecer también el código *CMake* que busca los recursos y crea las correspondientes variables. Esta tarea se sale de los objetivos de esta introducción a *CMake*.

A.5 CTest

Las herramientas *CTest* y *CDash* nos permiten gestionar el lanzamiento y la visualización de resultados de los tests asociados a nuestro proyecto. En esta sección incluimos una breve introducción a *CTest*.

Podemos trabajar con *CMake* para gestionar el proyecto sin entrar en nada referente a creación de tests. Si éste es su caso, puede ignorar esta sección. Sin embargo, es importante presentar algún ejemplo de cómo podemos resolverlo con *CTest*, no tanto porque esta herramienta sea especialmente importante, sino porque es fundamental entender que el desarrollo de grandes proyectos deberían incorporar siempre una parte de pruebas.

Suponga que trabaja en un proyecto donde varios programadores incorporan cambios al código fuente. Por ejemplo, podría ser un proyecto de software libre donde el código está en un repositorio compartido. Cuando un programador cambia algo, lo incorpora al servidor y todos los demás pueden actualizarlo. Puede haber decenas de programadores colaborando.

Imagine que considera conveniente cambiar un trozo de código de un archivo básico que se usa en muchos sitios. Si el cambio no es simple, es probable que se ponga nervioso a la hora de incorporar el nuevo código, porque si por casualidad se le ha olvidado un detalle, podría provocar que muchos programas fallen y que muchos de esos colaboradores se desesperen al ver los programas fallar.

Es conveniente hacer pruebas y confirmar que todo parece ir bien. Para ayudar en esa tarea, podemos crear programas específicamente para llevar a cabo tests. En principio, sólo sirven para los desarrolladores, pues son programas que se van a lanzar para confirmar que el código funciona.

A.5.1 Incorporar tests al proyecto

Para poder gestionar tests en nuestro proyecto, lo primero que vamos a hacer es activarlos. Para ello, comenzamos con la orden «enable_testing» en el archivo raíz antes de añadir tests. Como resultado, también se añade un nuevo objetivo «test» de forma que podemos lanzar las pruebas con «make».

Para añadir cada uno de los tests al proyecto vamos a usar la orden «add_test». Esta orden nos permite crear un nuevo test, darle un nombre e indicar el programa que hay que lanzar para llevarlo a cabo. El formato que usaremos es el siguiente:

```
add_test ( NAME <nombre> COMMAND <programa> [<parámetro1> ...] )
```

aunque si consulta el manual verá que hay otras posibilidades con más opciones⁸.

La forma en que se crea un test es realmente simple: le indicamos un programa con los parámetros asociados de forma que el entero que devuelve el programa indica si el test ha tenido éxito o no. Si devuelve un cero, el test se ha pasado; en caso contrario, ha fallado.

Para añadir un par de pruebas a nuestro proyecto *GEOM* creamos un par de programas, uno para calcular la longitud de una serie de puntos en un archivo y otro para calcular un rectángulo que los encierra (recuerde que eran operaciones que se realizaban en las aplicaciones de ejemplo).

En el árbol fuente añadimos un nuevo directorio «tests» donde añadir programas que corresponden a las pruebas que deseamos realizar:



El nuevo archivo «CmakeLists.txt» para ese directorio puede ser el siguiente:

```
1 # Aplicaciones tests de la biblioteca "formas"
2
3 add_executable ( test_longitud test_longitud.cpp )
4 target_link_libraries ( test_longitud formas )
5
6 add_executable ( test_bounding test_bounding.cpp )
7 target_link_libraries ( test_bounding formas )
```

Para incorporarlo al proyecto, basta con añadir una orden «add_subdirectory» en el padre. Con esto, la orden «make» creará también los dos programas que hemos añadido.

Estos programas están diseñados para usar funciones de la biblioteca y devolver el valor cero si todo se ha calculado bien. Por ejemplo, el programa «test_longitud.cpp» es el siguiente:

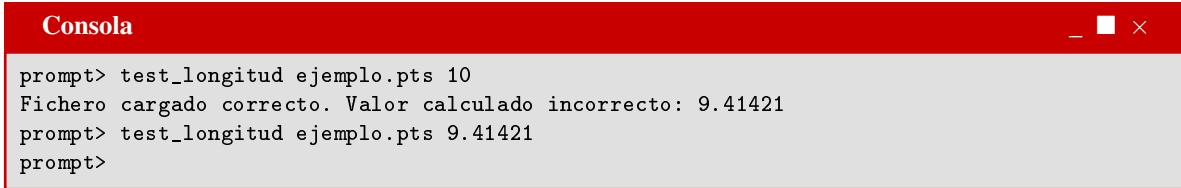
```
1 #include <iostream>
2 #include <fstream>
3 #include <cmath> // fabs, sqrt
4 #include "punto.h"
5 using namespace std;
6
7 int main(int argc, char* argv[])
8 {
9     if (argc!=3) {
10         cerr << "Uso: " << argv[0] << " <fichero puntos> <valor esperado>" << endl;
11         return 1;
12     }
13     else {
14         const char * fichero= argv[1];
15         double valor= atof(argv[2]);
16
17         ifstream f(fichero);
18         if (!f) {
19             cerr << "Error: no se abre " << argv[1] << endl;
20             return 2;
21         }
22
23         double l=0;
24         Punto anterior, siguiente;
25
26         if (Leer(f, anterior)) {
27             while (Leer(f, siguiente)) {
28                 l+= Distancia(anterior,siguiente);
29                 anterior=siguiente;
30             }
31         }
32
33         if (!f.eof()) {
34             cerr << "Error inesperado. No se ha leído toda la entrada" << endl;
35     }
```

⁸Realmente también se puede simplificar, pero como vamos a añadirlos en el directorio raíz tenemos que usar esta versión más elaborada

```

35         return 3;
36     }
37     if (fabs(l - valor) > 1e-4 ) {
38         cerr << "Fichero cargado correcto. Valor calculado incorrecto: "
39         << l << endl;
40         return 4;
41     }
42 }
43 }
44 }
```

Es un programa que recibe un archivo que contiene puntos y el valor que debería obtenerse como resultado de calcular la distancia acumulada entre cada dos puntos del archivo. Por ejemplo, se puede ejecutar:



```

prompt> test_longitud ejemplo.pts 10
Fichero cargado correcto. Valor calculado incorrecto: 9.41421
prompt> test_longitud ejemplo.pts 9.41421
prompt>
```

que ha devuelto un valor distinto de cero en el primer caso y cero en el segundo.

Para añadir el test al proyecto usamos la orden «`add_test`» de la siguiente forma:

```

1 add_test (NAME LongitudTrayecto COMMAND
2           test_longitud ${GEOM_SOURCE_DIR}/data/ejemplo.pts 9.41421 )
```

donde puede ver que hemos añadido un directorio «`data`» para incluir algunos archivos de ejemplo. Para que el programa pueda recibir este nombre, aprovechamos el valor de la variable que nos indica dónde están los fuentes.

De forma similar podemos añadir una prueba de esos mismos datos para obtener el rectángulo mínimo que los contiene. La línea para añadir este otro test es:

```

1 add_test (NAME BoundingBox COMMAND
2           test_bounding ${GEOM_SOURCE_DIR}/data/ejemplo.pts 0 0 5 5 )
```

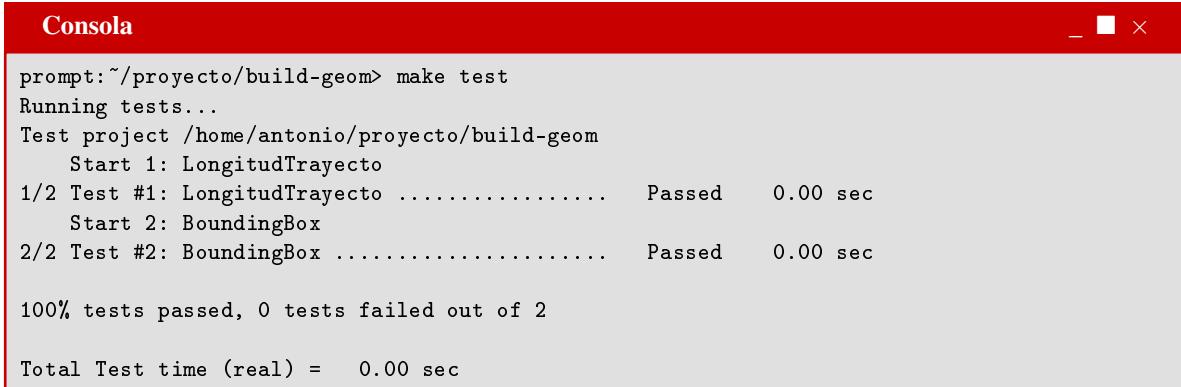
Observe que para ambos ejemplos hemos dado nombres a los tests evitando espacios y caracteres especiales. Como resultado, las últimas líneas del archivo «`CMakeLists.txt`» del directorio raíz serán las siguientes:

```

1 enable_testing()
2 add_test (NAME LongitudTrayecto COMMAND
3           test_longitud ${GEOM_SOURCE_DIR}/data/ejemplo.pts 9.41421 )
4 add_test (NAME BoundingBox COMMAND
5           test_bounding ${GEOM_SOURCE_DIR}/data/ejemplo.pts 0 0 5 5 )
```

A.5.2 Ejecución de tests: `ctest`

Una vez que hemos realizado todos los cambios indicados en la sección anterior, no tenemos más que volver a ejecutar «`make`» para que se creen todos los nuevos objetivos. Tras su ejecución, tenemos los programas y un nuevo objetivo para «`make`». Lanzamos las pruebas:



```

prompt:~/proyecto/build-geom> make test
Running tests...
Test project /home/antonio/proyecto/build-geom
  Start 1: LongitudTrayecto
    1/2 Test #1: LongitudTrayecto ..... Passed    0.00 sec
      Start 2: BoundingBox
    2/2 Test #2: BoundingBox ..... Passed    0.00 sec

  100% tests passed, 0 tests failed out of 2

  Total Test time (real) = 0.00 sec
```

El resultado es que se han lanzados todos los test y se ha dado un informe sobre qué ha pasado. En nuestro caso, las dos pruebas han tenido éxito.

Tenga en cuenta que estas pruebas estarán siempre disponibles. La forma de usarlas será ir modificando el software y conforme se incorporan nuevos cambios, volver a lanzarlas. Si hacemos un cambio que rompe el funcionamiento de alguna función de las que se usan —por ejemplo, la función `Distancia` entre dos puntos— el test que la usa probablemente dejará de funcionar —en nuestro ejemplo, la longitud—.

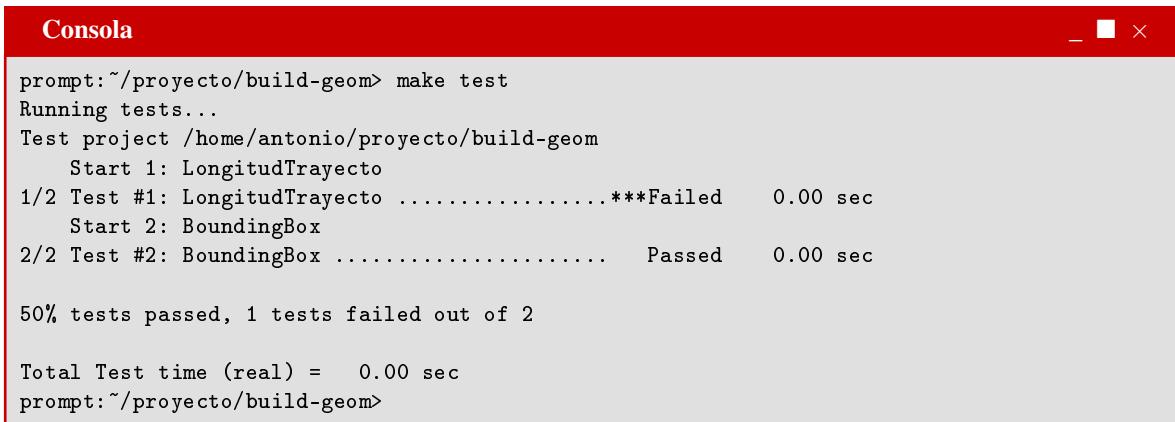
Este nuevo objetivo no es más que una forma indirecta de llamar al programa «`ctest`». Si prueba a ejecutarlo en el mismo directorio, verá que obtiene el mismo informe. Lo más interesante de usar directamente «`ctest`» es que podemos darle opciones para que realice otras acciones. Por ejemplo:

- «ctest -N»: lista los tests disponibles.
- «ctest -R <exp_reg>»: lanza test que coinciden con la expresión regular.
- «ctest -V»: lanza los test mostrando información de la orden y la salida obtenida.
- «ctest -I <índices>»: puede especificar los números de los tests a lanzar.
- etc.

Por ejemplo, imagine que el test de la longitud debería dar el valor 90 para ese archivo. La línea sería la siguiente:

```
1 add_test(NAME LongitudTrayecto COMMAND
2           test_longitud ${GEOM_SOURCE_DIR}/data/ejemplo.pts 90 )
```

Con este configuración, la ejecución de los tests sería, entre otras líneas:



```
prompt:~/proyecto/build-geom> make test
Running tests...
Test project /home/antonio/proyecto/build-geom
  Start 1: LongitudTrayecto
1/2 Test #1: LongitudTrayecto .....***Failed    0.00 sec
  Start 2: BoundingBox
2/2 Test #2: BoundingBox .....Passed    0.00 sec

50% tests passed, 1 tests failed out of 2

Total Test time (real) = 0.00 sec
prompt:~/proyecto/build-geom>
```

Podemos lanzar el test número 1 con la opción de que dé más información sobre lo que ejecuta y lo que escribe el programa. El resultado contendría, entre otros líneas:



```
prompt:~/proyecto/build-geom> ctest -V -I 1
test 1
  Start 1: LongitudTrayecto

  1: Test command: /home/antonio/proyecto/build-geom/formas/tests/test_longitud
     "/home/antonio/proyecto/geom/data/ejemplo.pts" "90"
  1: Test timeout computed to be: 9.99988e+06
  1: Fichero cargado correcto. Valor calculado incorrecto: 9.41421
  1/2 Test #1: LongitudTrayecto .....***Failed    0.00 sec
```

donde podemos ver exactamente la orden que se ha ejecutado y lo que ha escrito el programa.

```

#include <iostream>
#include <fecha.h>
using namespace std;

Fecha CentroPeriodo(Fecha const f1, Fecha const f2)
{
    return f1+(f2-f1)/2;
}

void Intercambia(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;
}

```

B Entornos integrados de desarrollo

Introducción	109
Utilidades que ofrece un IDE	
El entorno de desarrollo QtCreator	110
QtCreator y QMake/CMake	
Crear un proyecto	
Introducción a las opciones del entorno	
Compilación y depuración con QtCreator ..	119
Depuración	

B.1 Introducción

El desarrollo de proyectos complejos requiere de herramientas automáticas. Aunque en las primeras etapas de aprendizaje, pueda resultar pedagógico realizar las tareas de forma más o menos manual, en la práctica, no tiene sentido complicar innecesariamente muchas tareas que se pueden realizar automáticamente o, al menos, de una forma asistida que minimiza las posibilidades de error y el tiempo de trabajo.

Un entorno integrado de desarrollo —normalmente llamado **IDE**, un acrónimo del inglés *Integrated Development Environment*— es una aplicación diseñada para ayudar al programador en el desarrollo de software.

En un curso de metodología de la programación, cuando los proyectos tienen cierto tamaño, es cuando el estudiante puede empezar a sacar verdadero rendimiento a estos entornos. En este tema vamos a presentar algunos conceptos básicos para el uso de entornos de desarrollo, centrándonos especialmente en las utilidades más básicas que encontrará prácticamente en cualquiera de ellos. En concreto, vamos a presentar cómo usar «**QtCreator**», aunque muchos de los comentarios tendrían que repetirse de forma similar en otros programas.

Vamos a suponer que el lector tiene cerebro, así que no vamos a decir que **Run** ejecuta, que **Build** construye o que las opciones de edición están en **Edit**. La idea es recorrer rápidamente algunas opciones y algunos pasos para animar a que el estudiante pruebe el desarrollo con uno de estos entornos. Presentaremos las opciones más habituales de forma que sea fácil entender en qué podría consistir una sesión de trabajo con este entorno.

B.1.1 Utilidades que ofrece un IDE

El conjunto de herramientas que conforman un entorno de desarrollo no está definido claramente, puesto que siempre se puede diseñar para incluir nuevos módulos, especialmente si se ha creado para algún lenguaje o tipo de problemas con características especiales. Sin embargo, podemos comentar brevemente algunas de las habituales:

- **Editor de texto.** Para escribir un programa necesitamos un editor de texto. Normalmente, estará enriquecido para facilitar el manejo del programa:
 - Resaltado de la sintaxis. Destaca los elementos que componen la sintaxis del lenguaje para facilitar la lectura del programa. No sólo un resaltado básico de algunas palabras, sino incluyendo detalles más relacionados con la sintaxis o incluso la semántica del lenguaje. Por ejemplo, con avisos gráficos de estructuras que no van a compilar o que no están conforme a otra parte del código.
 - Ayuda en la edición con autocompletado. El editor sugiere posibles identificadores, por ejemplo, una lista de miembros para seleccionar un campo de una estructura o los parámetros que corresponden a la llamada de la función.
 - Facilita la navegación por el programa. Permite saltar fácilmente entre puntos del programa, no sólo seleccionando un nuevo archivo sino también atendiendo a conceptos más cercanos al lenguaje, como tipos de datos, objetos o características relacionadas con el punto de edición actual, etc..
- Herramientas para lanzar la *compilación*, *enlazado* y *ejecución* del programa. Un **IDE** puede incluir los programas para compilar y enlazar, aunque estrictamente lo podemos considerar independiente de éstos; existen algunos entornos que pueden adaptarse a distintos compiladores o incluso a distintos lenguajes; en la práctica, como paquete integrado, debería considerarse como parte del entorno de desarrollo. Para facilitar su uso, debería simplificar la selección de opciones y la generación automática de los archivos ejecutables.

- Herramientas para la *depuración del programa*. Un *IDE* debería facilitar la depuración del programa con el trazado de la ejecución, permitiendo la ejecución paso a paso, la ruptura de la ejecución, la visualización de datos, etc. Esta parte probablemente sea la más interesante, porque esta tarea puede llegar a ser muy complicada; si un error es difícil de localizar, el tiempo para hacerlo puede ser mucho más pequeño si dispone de un buen depurador.
- *Gestión de proyectos*. Aunque pueda resultar una característica de esperar, la ponemos explícitamente para recordar que un proyecto puede estar compuesto de múltiples módulos y recursos asociados. Además, pueden estar relacionados de forma que un proyecto explote las características desarrolladas en otro. Cada uno tendrá su configuración, sus módulos. Un *IDE* debe facilitar el manejo de proyectos; por ejemplo, crear una configuración propia para cada uno, facilitando que el programador pueda saltar de uno a otro fácilmente o incluso desarrollar varios proyectos a la vez.

Estas características básicas son las que vamos a visitar brevemente en este tema. Sin embargo, es conveniente puntualizar que los entornos de desarrollo modernos suelen incluir otras herramientas. Algunas de ellas pueden ser:

- *Asistentes* en la creación de nuevas componentes. Por ejemplo, un asistente para introducir un nombre de una clase y crear la plantilla de código que se necesita en la creación de un archivo de cabecera, con directivas de precompilador y algunas partes de la sintaxis de la clase ya introducidas.
- *Herramientas de análisis de rendimiento*. Por ejemplo, evaluar qué tiempo se ha necesitado en cada componente para centrar los esfuerzos de rediseño en los más importantes.
- *Herramientas de diseño visual de interfaces*. Facilitan incluir componentes como etiquetas, botones, menús, etc. sin necesidad de escribir código, editando la interfaz gráfica que generará automáticamente el código relacionado.
- *Herramientas para control de versiones*. Por ejemplo, incluyendo opciones para intercambiar información con un repositorio que contiene el proyecto.
- *Herramientas gráficas de diseño de software*, por ejemplo, para presentar clases, objetos, relaciones, etc.

Este tema, de *introducción* a los entornos de desarrollo, no abordará estos componentes. Si quiere más detalles, consulte los manuales del entorno que está usando.

B.2 El entorno de desarrollo QtCreator

Este entorno se creó como parte de las herramientas asociadas a las bibliotecas «**Qt**». Inicialmente se desarrollaron para poder crear interfaces gráficas con C++ a lo largo de los años 90. Aunque se basó en este lenguaje, desarrolló sus propias herramientas¹ para facilitar algunas extensiones del lenguaje.

El proyecto ha seguido creciendo. Actualmente es un entorno de desarrollo que incluye muchas más cosas que las que vamos a ver aquí. No sólo en forma de bibliotecas, sino también con un nuevo lenguaje *QML* para desarrollo rápido que se integra con código *JavaScript* y C++.

A pesar de su complejidad, el entorno *QtCreator* es una buena herramienta incluso en un curso básico de programación. Se pueden desarrollar programas simples con la ventaja de un *IDE* estable, moderno y que puede instalarse en GNU/Linux, Mac OS X o Microsoft Windows.

B.2.1 QtCreator y QMake/CMake

La gestión de los proyectos con *QtCreator* se realiza con un tipo concreto de archivos que son interpretados con el programa **qmake**, específicamente diseñado para proyectos con la *Qt*. Básicamente, este programa es capaz de generar archivos **Makefile** asociados al proyecto para que el compilador de nuestro sistema pueda crear los ejecutables. Es una herramienta que permite, por tanto, la compilación en múltiples sistemas operativos y con distintos compiladores.

Este programa recuerda a *CMake*, que se introduce en el tema A, en la página 89. Esta herramienta puede considerarse más genérica que *qmake*, puesto que no está asociada a *Qt*, puede encontrarla en otros proyectos, incluso puede ser la solución para gestionar proyectos en otros entornos de desarrollo.

Si va a desarrollar otros proyectos que no tienen relación con *Qt*, la gestión con *CMake* puede ser una buena opción. Si a esto unimos que el entorno *QtCreator* permite gestionar la solución también con *CMake*, en este curso proponemos trabajar con este tandem para nuestros proyectos software.

En cualquier caso, si quiere gestionar un proyecto con **Qmake**, no es un modelo muy distinto, aunque lógicamente **QtCreator** está mejor adaptado a él. Los archivos de configuración del proyecto tendrán extensión «.pro», y la sintaxis que aparece es distinta; por ejemplo, en una aplicación de consola, se incluirá algún detalle para indicar que es una aplicación de consola que evita la *Qt*. Si le interesa estudiar estas bibliotecas en más profundidad, podría ser una mejor alternativa.

Recuerde que si el programa contiene código C++ estándar, cuando termine el proyecto, podrá empaquetarlo y llevarlo a otros sistemas para compilarlo fácilmente con disponer sólo de *cmake* y el correspondiente compilador de C++.

QMake/CMake vs make

Las herramientas **QMake/CMake** son de un nivel de abstracción superior al programa clásico **make**. Están desarrolladas para poder gestionar un proyecto software con una sintaxis y opciones más simples. La ventaja es que podemos describir fácilmente nuestro proyecto y llevárselo a distintos sistemas.

Son *meta-generadores*, un generadores de archivos generadores. Con estos programas y su correspondiente archivo de configuración, se pueden obtener los archivos que permiten generar los objetivos finales del proyecto. Por ejemplo, se pueden

¹Si tiene curiosidad, busque *Meta-Object Compiler* —moc— en internet.

obtener los archivos **Makefile** —que luego se procesarán con la orden **make**— o los archivos de configuración para **Visual Studio**, que luego se podrán abrir o procesar con el correspondiente programa.

Cuando creamos un nuevo proyecto, tendremos que seleccionar qué tipo de herramienta vamos a usar para gestionarlo. En nuestro ejemplos, usaremos **QMake** o **CMake**, aunque también se incluye **Qbs**. Una vez que el proyecto está abierto y trabajamos con el entorno, éste se encargará de realizar la compilación y lanzar el programa cuando usemos los botones correspondientes.

Veamos unos ejemplos de estos formatos con un pequeño proyecto para que compruebe que la sintaxis, al menos para casos simples, puede llegar a resultar bastante simple. Suponga un programa que comprueba el funcionamiento del algoritmo de ordenación *ShellSort*, para lo que creamos dos módulos:

- Un módulo que implemente el algoritmo. Tiene dos archivos: **ordenar.h** y **ordenar.cpp** para ofrecer la función de ordenación correspondiente.
- Un módulo con la función **main**. Crea un vector con valores enteros aleatorios con un tamaño fijado por el usuario en tiempo de ejecución, lo ordena y muestra el resultado.

Una versión simple de proyecto-QtCreator

El proyecto lo puede gestionar con un archivo **shellsort.pro** específico de **QtCreator** con las siguientes entradas:

```
TEMPLATE = app
CONFIG += console c++11
CONFIG -= app_bundle
CONFIG -= qt

SOURCES += main.cpp \
    ordenar.cpp

HEADERS += \
    ordenar.h
```

donde no hemos indicado que queremos un archivo ejecutable. Se entiende que el objetivo del proyecto es crear el ejecutable **shellsort**, el nombre del proyecto, aunque se podría cambiar explícitamente.

Este archivo podría ser el que genera automáticamente **QtCreator** al crear un proyecto con ese nombre y llamar a añadir un archivo de cabecera **ordenar.h** y un archivo fuente **ordenar.cpp**.

Una versión simple de **CMakeLists.txt**

Puede usar el tema A para esta introducción, más concretamente la sección A.2 —página 90— aunque incluimos una pequeña reseña para hacer este tema más autocontenido.

Tenga en cuenta que aunque usemos **CMake** para gestionar los proyectos, no supondremos un conocimiento profundo sobre esta herramienta. Nuestros programas serán relativamente sencillos por lo que sólo tenemos que entender cómo se generan los programas a partir de un archivo **CMakeLists.txt**, así como conocer algo sobre su sintaxis para añadir o eliminar archivos del proyecto.

Para nuestro ejemplo de ordenación podemos crear el siguiente archivo **CMakeLists.txt**:

```
1 project( shellsort )
2 cmake_minimum_required( VERSION 3.0 )
3 add_executable( shellsort ordenar.h ordenar.cpp main.cpp )
```

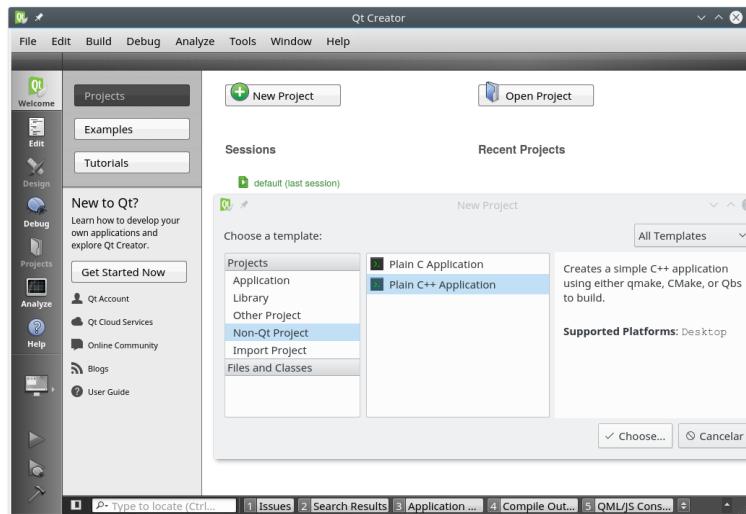
En principio, la diferencia más importante con respecto a los proyectos de **QMake** es que **QtCreator** tiene que llamar a una herramienta de terceros —el ejecutable **cmake**— para traducir los archivos **CMakeLists.txt** a archivos **Makefile**. La mayoría de las veces lo hará automáticamente, aunque el entorno incluye la opción para que el usuario pueda lanzarlo explícitamente.

En este pequeño ejemplo es fácil ver en qué consiste el proyecto. Tal vez la parte más interesante y que puede sorprender es que la lista de archivos fuente que componen el ejecutable puede contener archivos cabecera. Esto no es un inconveniente, pues se entiende que éstos no se compilan. Podríamos eliminarlo, pero es conveniente ponerlos explícitamente para que el entorno sepa que son parte del proyecto.

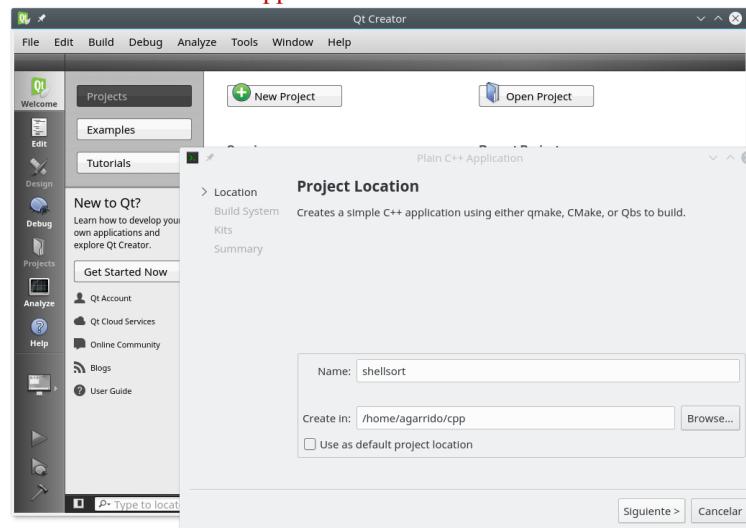
B.2.2 Crear un proyecto

Para tomar contacto con este *IDE*, vamos a crear un proyecto con el asistente del propio entorno, aunque más adelante veremos que podríamos hacerlo simplemente creando los archivos de configuración que hemos visto. El asistente comienza con el botón *New Project* y nos pregunta sucesivamente las distintas opciones hasta obtener una plantilla inicial con un simple archivo **main.cpp**.

Para este curso de C++, los proyectos se configuran para la ejecución en consola, sin entorno gráfico. La configuración, por tanto, la comenzamos seleccionando estas opciones:

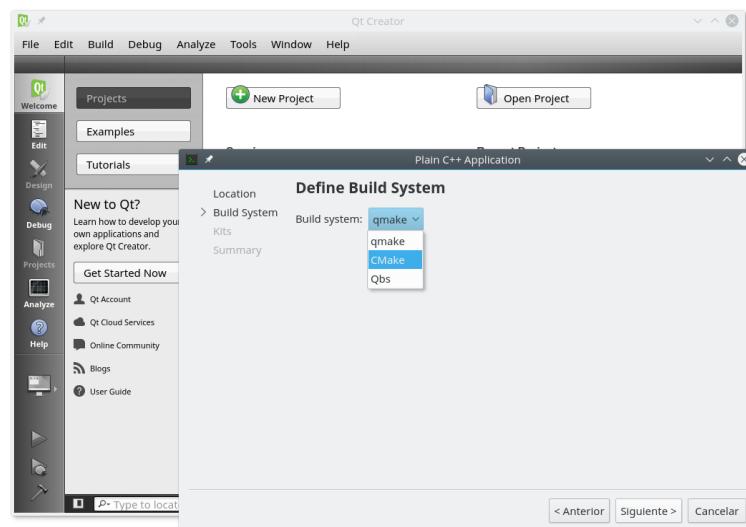


Después tendrá que darle un nombre y seleccionar un directorio donde almacenarlo. Suponga que para nuestro ejemplo el nombre es **shellsort** y lo creamos en el directorio **cpp**:



El resultado de estas opciones es que el entorno crea un directorio con el nombre del proyecto como subdirectorio de **cpp**. Es decir, podríamos seleccionar este mismo directorio para otros proyectos sin que resultara problemático.

El siguiente paso es la selección del sistema de gestión de nuestro proyecto. Como hemos indicado anteriormente, tenemos tres opciones. Dependiendo de lo que escogamos, el asistente finalizará de distinta forma. En este paso, seleccione una de las dos primeras opciones:

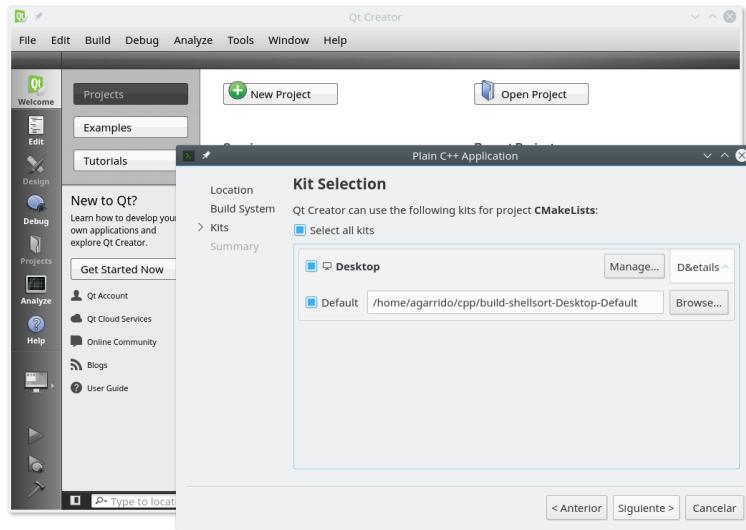


Después de este paso las ventanas que aparecen son distintas dependiendo de si ha seleccionado **CMake** o **QMake**. Describimos cómo finaliza el asistente, en cada caso, en las siguientes secciones.

Finalizando con CMake

En el caso de que hayamos seleccionado **CMake** y suponiendo que tenemos los programas asociados a éste en el sistema, el asistente nos indicará que hemos seleccionado el «**kit**» correspondiente.

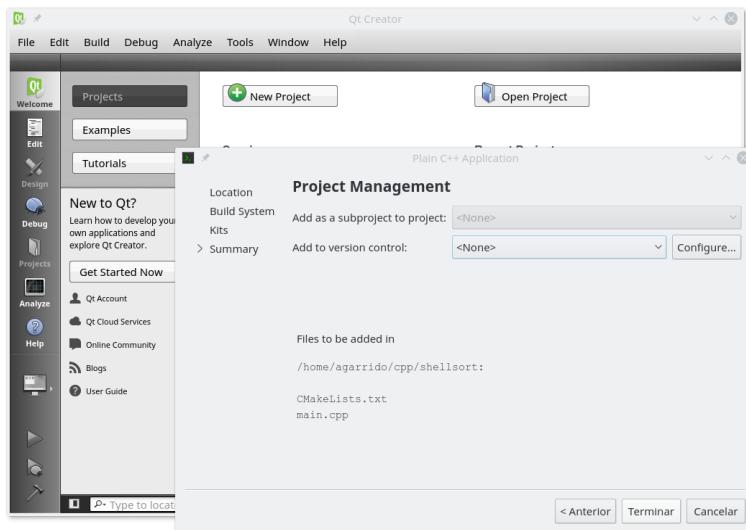
Un «**kit**» no es más que una forma de empaquetar las opciones que especifican una forma de generar y ejecutar aplicaciones para una determinada plataforma. Entre estas opciones se encuentra el compilador o herramientas asociadas. En nuestro caso, obtenemos:



donde hemos indicado que muestre los detalles pulsando en el botón *Details*.

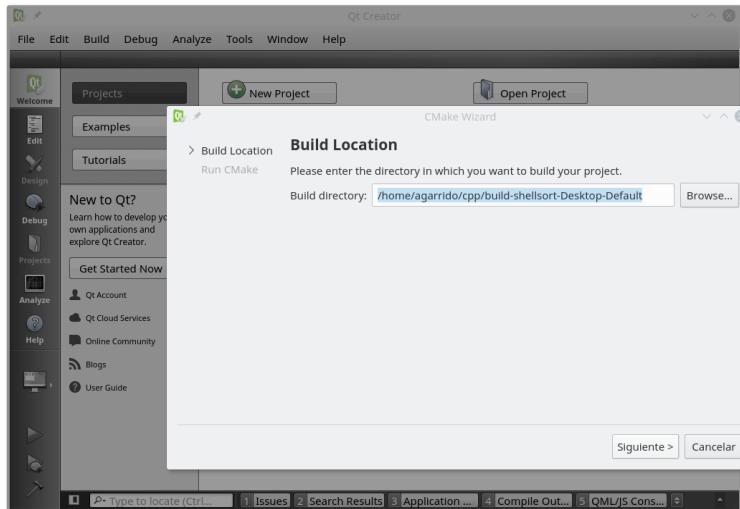
En esta ventana, *QtCreator* nos permite configurar dónde se generarán los archivos binarios. Note que el directorio es un subdirectorio de **cpp**, es decir, es “*hermano*” del que contiene los fuentes. Es un buen lugar para poder localizar dónde están los dos árboles asociados a nuestra aplicación.

En la última parte nos muestra la ventana final antes de crear el proyecto. En esta ventana nos informa de los archivos que va a generar y nos permite seleccionar un control de versiones. Si tiene instalado **git**, podría seleccionarlo, lo que implicaría un nuevo archivo automático en el proyecto —de nombre «**.gitignore**»— asociado a la gestión de proyectos con **git**. Pruebe a cambiarlo para ver que la lista inferior cambia. En nuestro ejemplo, ignoramos esa opción y pulsamos *Terminar*.



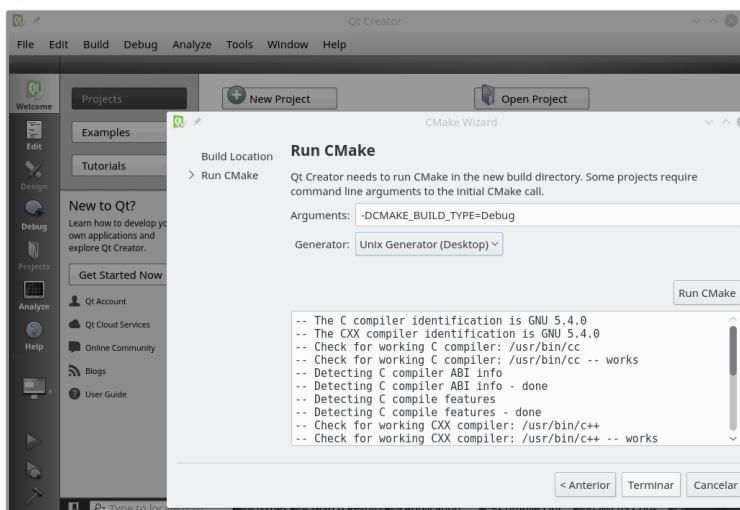
No, no es la última parte. Realmente falta algo más, puesto que la gestión con **CMake** implica la ejecución de **cmake** para crear los archivos **makefile**. Este es un detalle relevante: *QtCreator* “no ve” que un archivo está en el proyecto hasta que **cmake** ha procesado el archivo **CMakeLists.txt** y ha localizado que ese código será parte de la aplicación.

Como consecuencia, antes de terminar, tendremos que indicar al entorno que queremos lanzar `cmake` y hacerlo. En concreto, comienza con el directorio donde se lanzará:

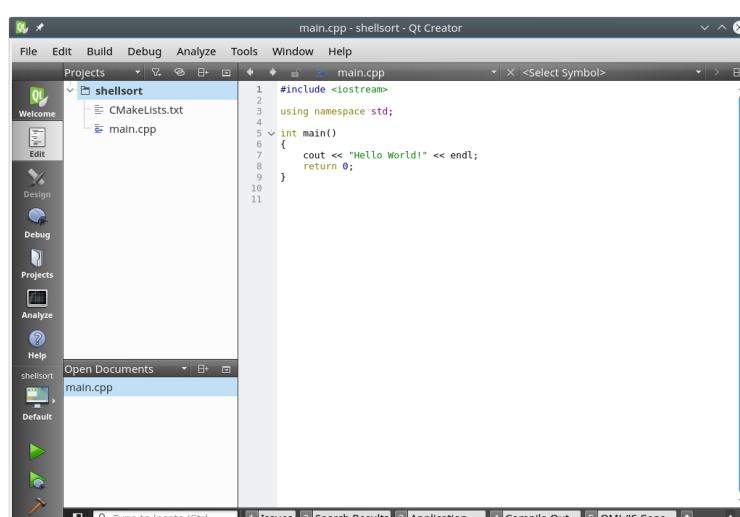


que coincide con lo que habíamos indicado anteriormente. Simplemente aceptamos el lugar y damos a siguiente para que nos presente la ventana de lanzamiento de `cmake`.

Esta ventana contiene un botón «Run CMake» que genera en el directorio anterior los archivos de configuración. Antes de ello, nos presenta una línea **Arguments** por si queremos indicar alguna opción a `cmake`. No es necesario escribir nada, pero nosotros indicamos explícitamente que la compilación incluirá información de depurado de forma que el proyecto siempre estará listo para lanzar el depurador. Tras pulsar el botón:



ya podemos terminar el proyecto para obtener la ventana con el código inicial: un programa del tipo “*Hola mundo*”.



Observe que también está el archivo **CMakeLists.txt** para configurar el proyecto. Puede modificarlo directamente, si lo desea. Si consulta lo que ha incluido el entorno encontrará algo como lo siguiente:

```
1 project(shellsort)
2 cmake_minimum_required( VERSION 2.8 )
3 aux_source_directory( . SRC_LIST)
4 add_executable(${PROJECT_NAME} ${SRC_LIST})
```

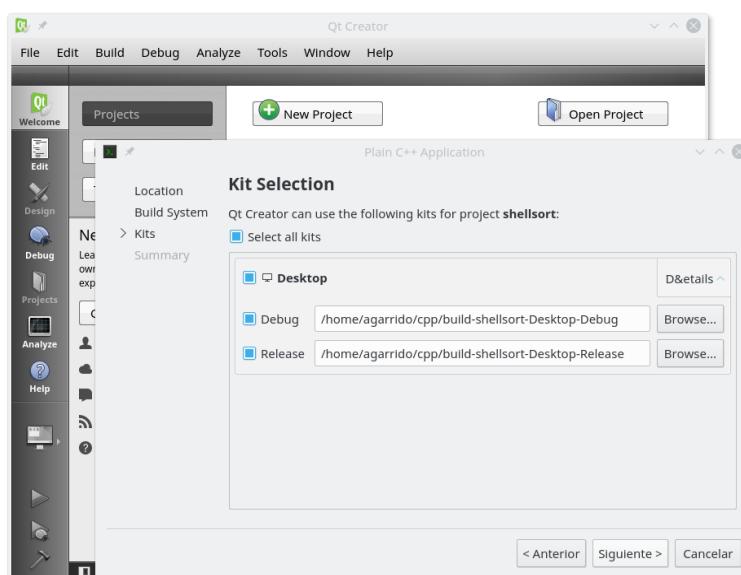
Este archivo es extremadamente simple. Ha creado la versión más básica para indicar que hay un proyecto con los fuentes que se incluyen en el directorio:

- En la línea 3 la orden **aux_source_directory** tiene un primer parámetro —un directorio— desde el que cargar los contenidos de la variable **SRC_LIST**. Es decir, asigna a ésta la lista de todos los fuentes de ese directorio.
- En la línea 4 la orden **add_executable** indica que el ejecutable tiene el mismo nombre del proyecto —**shellsort** como indica la línea 1— y que se compone con todos los fuentes anteriores.

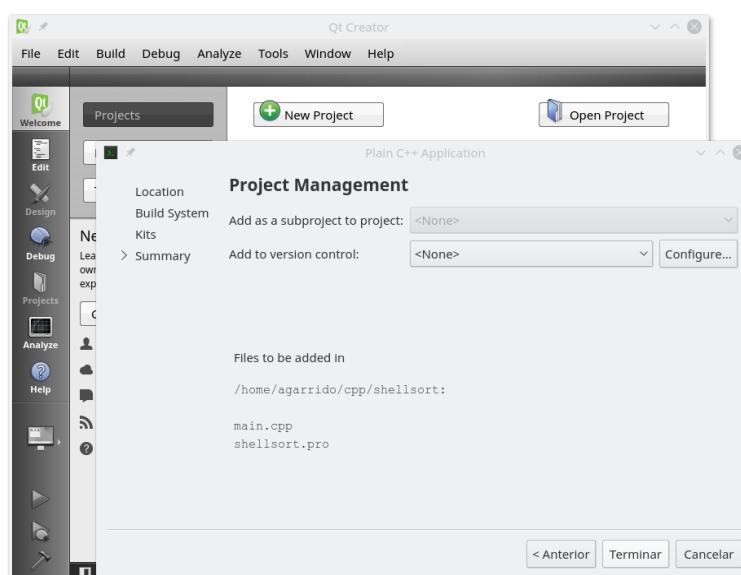
Es posible que más adelante quiera cambiar este archivo. Por ejemplo, con unas líneas más precisas como las que hemos presentado anteriormente. De esa forma, podrá indicar exactamente qué archivos están o no en esa lista.

Finalizando con QMake

En el caso de que hayamos seleccionado **QMake** la finalización es más simple, probablemente porque la gestión de este programa ya está integrada con el entorno. En este caso es interesante observar las opciones que aparecen con este «kit»:

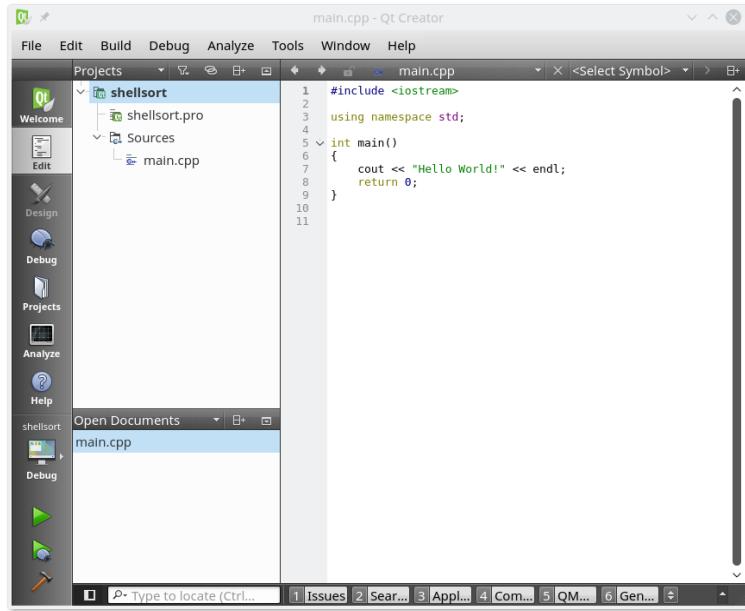


El «kit» configura dos posibles opciones. Para ello, establece dos directorios para binarios, uno para depuración y otro para versión final. Si deja seleccionados los dos, después podrá cambiar de uno a otro según lo que desee. Si acepta las opciones por defecto y sigue, obtendrá algo similar a lo que hemos visto, un resumen donde puede seleccionar si desea un control de versiones:



Note que con esta ventana podemos terminar la configuración. En los archivos que se generan se incluye ahora el archivo **shellsort.pro** que corresponde a la configuración con **QMake**.

Terminar implica directamente volver a la ventana del entorno para comenzar a trabajar. Puede consultar el contenido del archivo de configuración, donde podrá ver algo similar a lo que se ha comentado anteriormente.



B.2.3 Introducción a las opciones del entorno

No vamos a revisar todas las posibilidades del entorno, sólo las más relevantes y las que usaremos para las aplicaciones de consola. Las primeras opciones se obtienen en la ventana de “*bienvenida*” que se abre cuando lo lanzamos. En la figura B.1 se muestra ésta junto con una pequeña descripción de algunos botones.

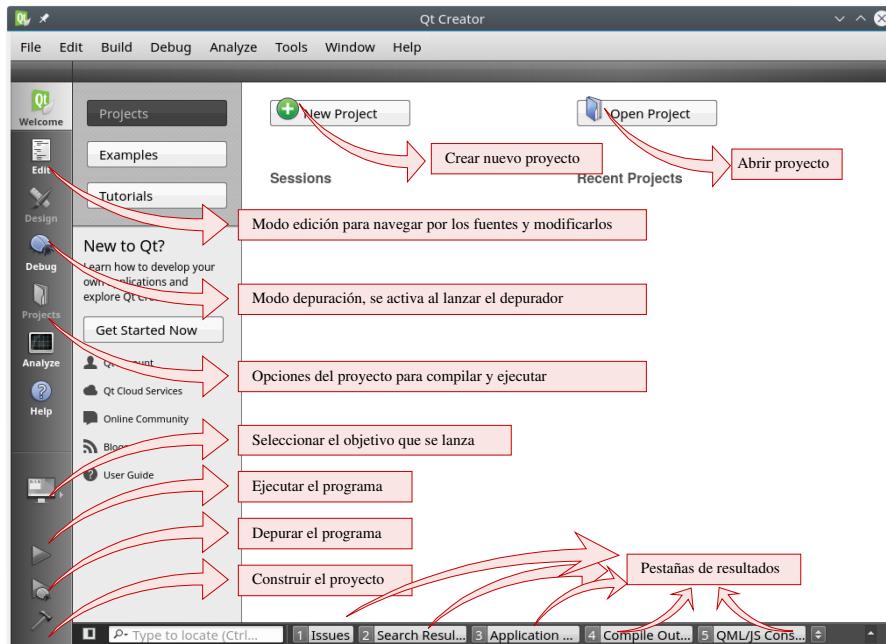


Figura B.1
Ventana inicial de QtCreator.

Observe que algunas de estas opciones no las vamos a usar por ahora. En concreto la de *Diseñar* y *Analizar*. La primera está desactivada en una aplicación de consola y la segunda de análisis de rendimiento que no vamos a necesitar.

Abrir un proyecto CMake

Para revisar con más detalle el entorno, vamos a abrir un proyecto que ya existe. Además, vamos a suponer que está ya configurado con **cmake** y lo abrimos por primera vez. Para ello, sólo tenemos que seleccionar el botón “**Open Project**” y navegar para localizar el archivo **CMakeLists.txt**. Este archivo ya lo tenemos con el siguiente contenido:

```

1 project ( SHELLSORT )
2 cmake_minimum_required (VERSION 3.0)
3
4 # Este if no es necesario, pero así aprovechamos las opciones de la GNU
5 if ( CMAKE_COMPILER_IS_GNUCXX )
6   message ( "Especificamos flags concretos para la gnu..." )
7   set ( SHELLSORT_EXTRA_CMAKE_CXX_FLAGS "-Wall -Wextra -pedantic -std=c++14"
8         CACHE STRING "Flags extra que se añaden a CMAKE_CXX_FLAGS" )
9   set ( CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${SHELLSORT_EXTRA_CMAKE_CXX_FLAGS}" )
10 )
11 else() # Si estamos con otro, al menos fijamos que es C++14
12   set ( CMAKE_CXX_STANDARD 14 )
13 endif( CMAKE_COMPILER_IS_GNUCXX )

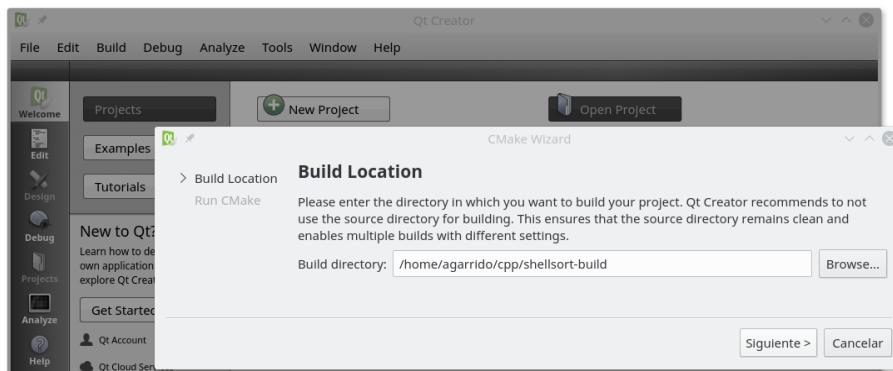
14 # El desarrollo se hace como Debug. Por comodidad, lo fijamos.
15 set (CMAKE_BUILD_TYPE "Debug" CACHE STRING "Fijamos Debug" FORCE)
16
17 # Usamos una variable para ir añadiendo archivos del proyecto
18 set ( SHELLSORT_SRCS
19       ordenar.h ordenar.cpp
20       main.cpp
21     )
22
23 # En ${SHELLSORT_SRCS} también están los .h
24 add_executable( shellsort ${SHELLSORT_SRCS} )

```

junto con los tres archivos que componen el programa. Note que hemos enfatizado que los archivos cabecera también están en la lista de fuentes —vea línea 19— y se listan en la última línea, donde se especifica cómo obtener el ejecutable.

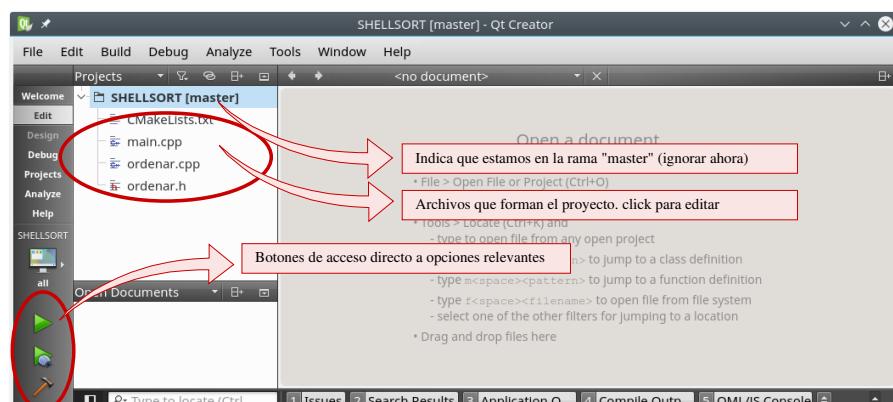
El archivo se encuentra en el directorio `cpp`, tras seleccionarlo el usuario tiene que:

- Indicar dónde quiere construir los ejecutables. Podemos dejar el directorio por defecto, que está en el mismo sitio que el proyecto, como subdirectorio de `cpp`.



- Ejecutar `cmake` antes de cargar el proyecto. El paso es similar a como indicábamos antes, aunque ahora nos podemos ahorrar añadir el argumento con el que se ejecutará `cmake`, pues ya está configurado en el archivo `CMakeLists.txt` —en la línea 15— el valor del tipo de construcción como `Debug`.

El resultado es que el entorno queda abierto con el proyecto para que el usuario comience a trabajar con él. Las opciones que vería son:



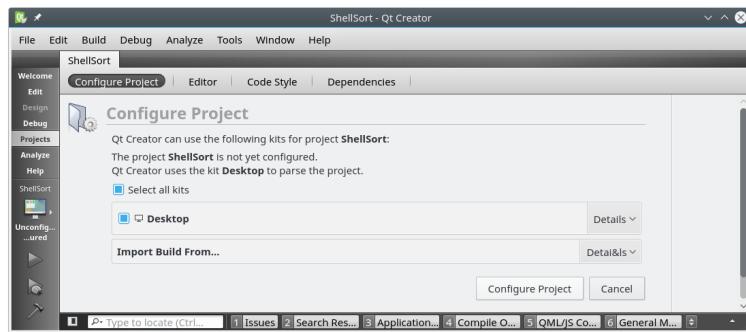
donde es interesante que se fije en que los archivos están en la ventana de proyectos y que los botones para ejecutar, depurar o construir están activados para su uso.

El detalle “**master**” en el nombre del proyecto es consecuencia de que el directorio que he abierto contenía los archivos de configuración de un repositorio para el control de versiones con «**Git**». Si no fuera así, simplemente no aparecería esa etiqueta. En principio, ignore esa información, pues no es parte de este tema (en el tema C, en la página 123 se explica más extensamente esta herramienta).

La próxima vez que abra el entorno, el proyecto le aparecerá en la lista de proyectos recientes. Seleccionándolo, pasará directamente a esta última ventana, pues el entorno ya ha almacenado la información asociada necesaria para **QtCreator**. Puede consultar el directorio del proyecto para encontrar el archivo **CMakeLists.txt.user** que se genera y gestiona automáticamente por el IDE.

Abrir un proyecto QMake

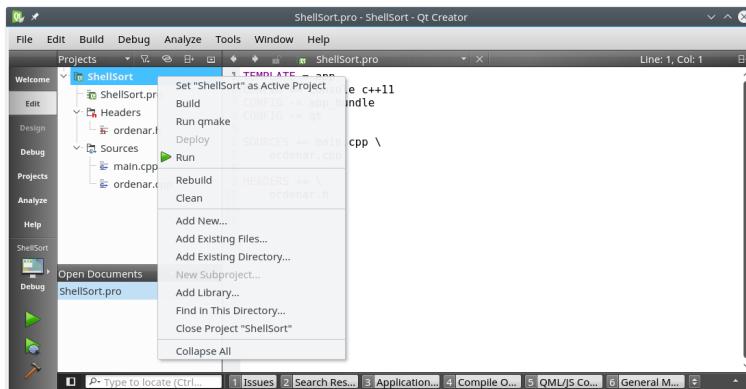
En este caso los pasos son similares, buscando el archivo con extensión **.pro** que le corresponde. Si el proyecto no se ha abierto antes, es posible que le aparezca una ventana para configurarlo. Recuerde que también debe seleccionar los directorios donde se van a generar los archivos compilados:



Si pulsa en configurar el proyecto, obtendrá una ventana similar a la anterior, aunque gestionando el proyecto con **QMake**. La presentación, por ejemplo, será distinta, pues mostrará los archivos cabecera independientemente de los archivos **cpp**, como puede ver en la siguiente figura.

Opciones del proyecto

Puede recorrer los distintos menús para ver la variedad de alternativas que tiene; en este caso, presentamos las opciones del menú que se obtiene con el botón derecho sobre el nombre del proyecto:



En esta ventana se puede ver que el proyecto está compuestos por archivos cabecera y fuentes presentados de forma independiente. El primer archivo contiene el proyecto. Además, no está bajo **Git**; no hay ninguna referencia a la rama *máster* de éste.

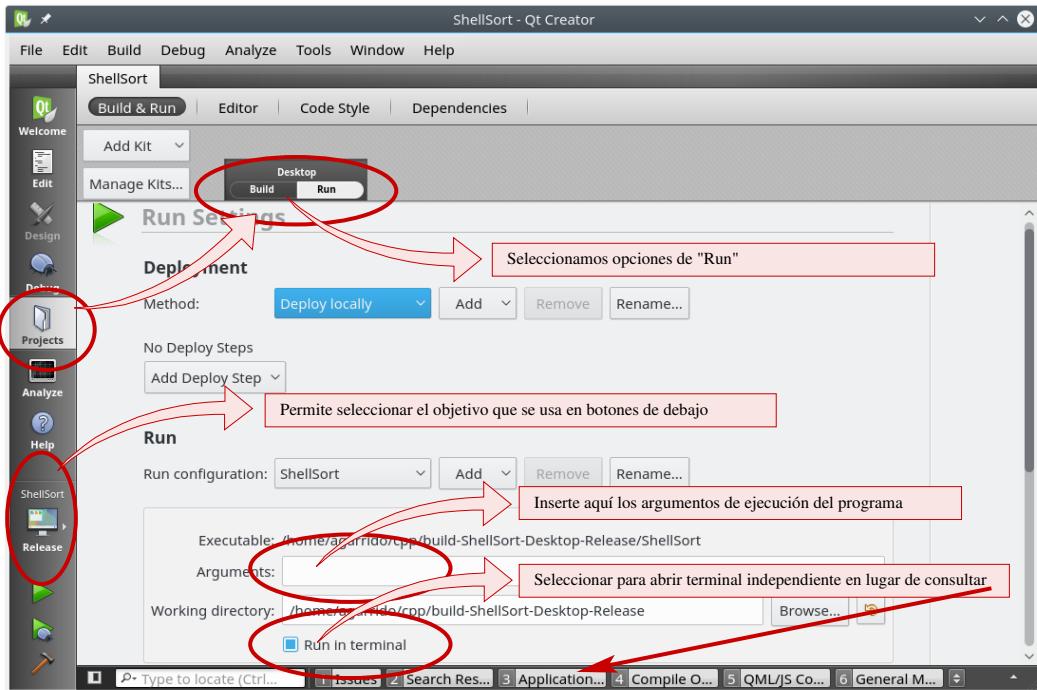
En este menú se pueden ver algunas opciones relevantes. Observe que se puede construir, ejecutar, limpiar, etc. La opción “**Run qmake**” está disponible como **Run cmake** si el proyecto está configurado para éste.

La parte que nos interesa enfatizar aquí es la posibilidad de añadir nuevas clases, ficheros, etc. Nos da acceso a un asistente que nos facilita incorporarlos al proyecto. Si lo hace así, el mismo entorno gestionará automáticamente el contenido del archivo **ShellSort.pro**.

En general, las opciones de gestión con **qmake** están mejor integradas que con **cmake** —no podría ser de otra forma— por lo que es habitual que algunas operaciones requieran una intervención más manual cuando se trata de éste. Por ejemplo, para incluir un nuevo archivo, lo más sencillo con **cmake** es modificar el archivo de configuración² y volver a ejecutar **cmake** para que se actualice el árbol del proyecto.

²Recuerde que la configuración generada automáticamente incluía todos los archivos fuente. Nosotros los escribiremos explícitamente para evitar que haya archivos no deseados.

Por otro lado, es necesario revisar alguna de las opciones del proyecto que podrá cambiar al pasar al modo **Projects** en la columna de la izquierda. En este caso puede cambiar opciones relacionadas con la construcción —**Build**— o ejecución —**Run**— actuales. Puede verlo en:

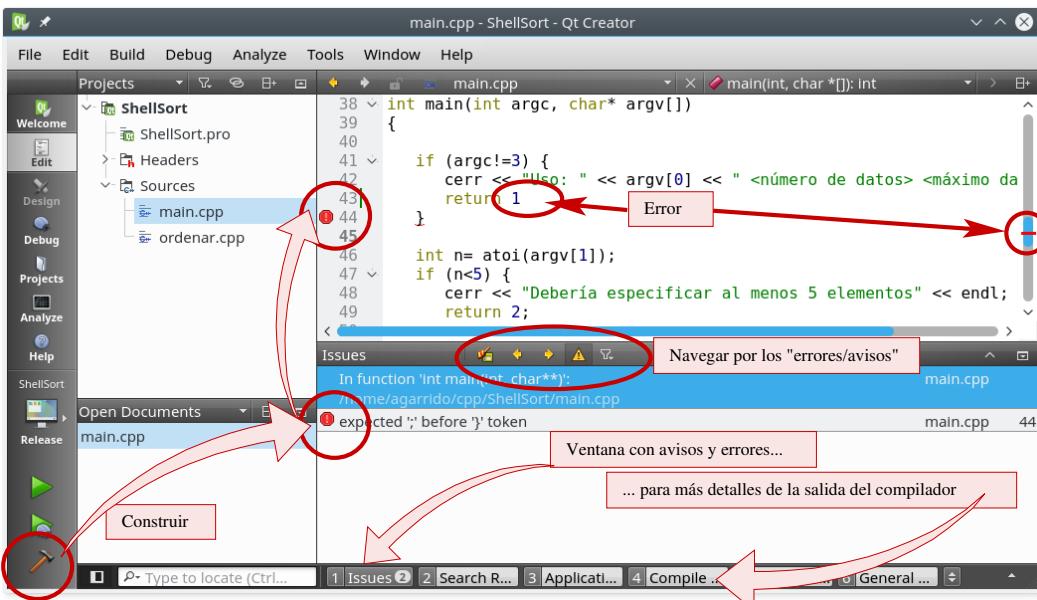


En nuestros problemas —que muchas veces tendrán una interfaz de consola— resulta especialmente de interés conocer que se pueden cambiar los argumentos de ejecución así como lanzar un terminal externo donde se ejecutará el programa.

Observe que hemos resaltado la selección del objetivo que hay encima de los tres últimos botones de la izquierda. En el caso de este proyecto con `qmake` podemos seleccionar entre **Debug** y **Release** para ejecutar la versión de depurado o definitiva. También se pueden configurar varios objetivos, por ejemplo, varios ejecutables con distintas opciones. En este caso, tendremos que seleccionar aquí el objetivo que nos interesa y lanzarlo o depurarlo con los botones que hay debajo. No vamos a entrar en detalles de cómo se hace, aunque veremos algún ejemplo práctico.

B.3 Compilación y depuración con QtCreator

Una de las ventajas de los *IDEs* es la facilidad para compilar y navegar por los errores de compilación. En nuestro entorno, pulsando en el botón inferior obtenemos una lista de los errores en la pestaña “**Issues**”. En la siguiente figura hemos provocado un error eliminando el carácter ‘;’ de la línea 33:



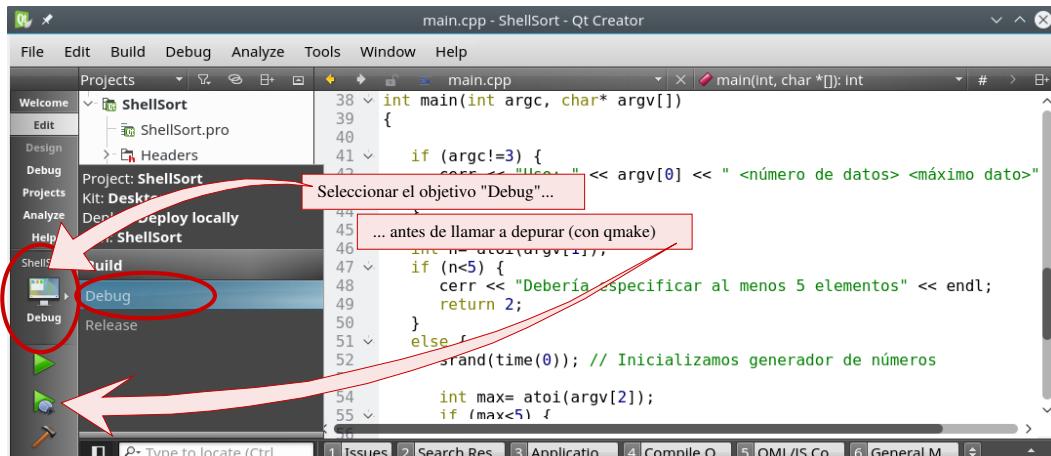
Al pinchar en el problema —en la pestaña “**Issues**”— directamente nos movemos a la línea con error, donde nos podemos ocupar de solucionarlo. Además, el error está descrito de forma simple, aunque la salida —que se puede consultar en la

pestaña **Compile Output**— sea mucho más larga. El salto tan rápido por cada uno de los puntos nos permite arreglar un gran número de errores en muy poco tiempo, recompilando rápidamente con un sólo “click” de ratón.

B.3.1 Depuración

Una vez que el programa está compilado, podemos ejecutarlo con el botón **Run** —el triángulo verde— que lanza el programa para obtener la salida en la pestaña **Application Output**—la número 3 en la figura— o en una terminal externa si lo hemos configurado en el proyecto.

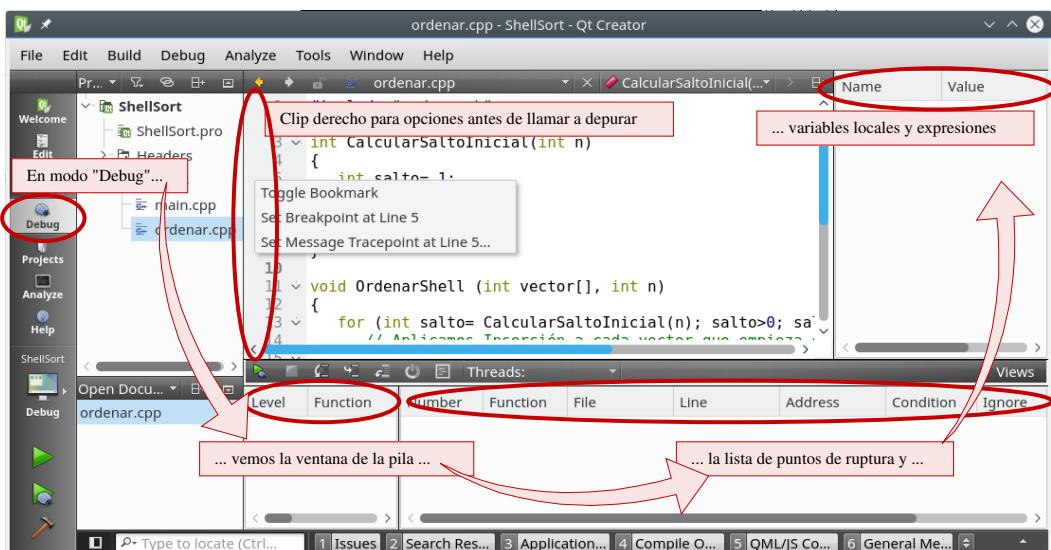
En caso de necesitar depurarlo, podemos pulsar el botón de depuración si el programa está compilado para ello. Tenga en cuenta que en **qmake** habíamos generado un objetivo explícitamente para depurar. Puede seleccionarlo antes de depurar:



mientras que si lo hacemos con **cmake**, donde sólo hemos creado un objetivo, éste debe haberse creado con las opciones de depuración.

El modo de depuración

En la columna de la izquierda disponemos de un botón para activar el modo de depuración. Realmente no lo tenemos que activar para comenzar una sesión de depuración, pues ésta normalmente comienza lanzando la ejecución con el botón de más abajo, el del “bicho”. Cuando la ejecución se detiene mientras trazamos, el entorno se muestra en este modo. Probablemente, el botón le será especialmente útil cambiando entre distintos modos, por ejemplo de “edición/depuración”. Cuando activamos este modo obtenemos algo como sigue:



donde aún no estamos trazando el programa. Simplemente se muestran las ventanas en su situación por defecto. En esta ventana, es interesante observar que:

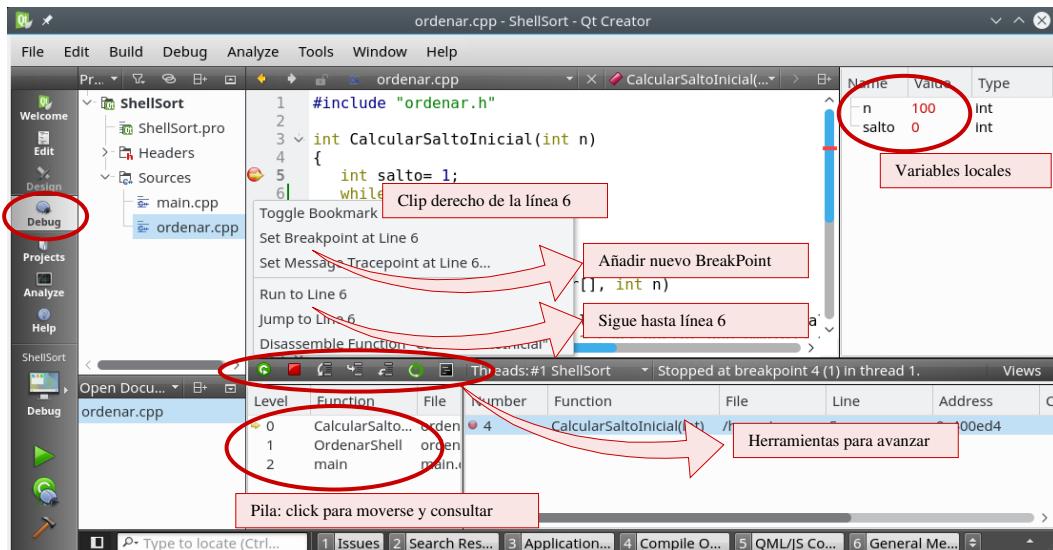
- Podemos pulsar con el botón derecho en cualquier línea del programa para obtener un menú. En éste, es de especial interés la segunda opción —vea la figura— que nos permite situar un punto de ruptura.
- El botón del triángulo con “el bicho” está activo y nos permite lanzar el programa para depurar.

Depurando

Para comenzar una sesión de depuración, lanzaremos el programa haciendo que se detenga para trazar qué está ocurriendo. Esta ejecución se puede hacer con el botón de depurado —sí, el del “bicho”—. Nuestro objetivo es que el programa se detenga y veamos qué está pasando. Esta parada normalmente la haremos:

- Sin indicar ningún lugar concreto; el programa se detiene con un error de ejecución inesperado. La ejecución del programa con una terminación anormal dejará el depurador con el programa finalizado pero con el mapa de memoria del momento del error. Por ejemplo, si ponemos una condición con **assert** y el programa falla en la condición, la primera opción es ejecutar en modo depuración para que se detenga y podamos consultar los detalles del error.
- Configurando un punto de parada. Podemos situar un punto de ruptura en algún lugar de ejecución para que el programa se detenga al llegar ahí. De esa forma estaremos en disposición de ir avanzando poco a poco y estudiando el comportamiento del programa.

En la siguiente figura se muestra cómo aparece el entorno después de lanzar el depurado con un punto de ruptura situado en la línea 5 que corresponde a la función *CalcularSaltoInicial*.



En esta figura se han enfatizado algunas de las opciones más usadas cuando depuramos:

- Podemos situarnos en otra línea —en la figura la línea 6— donde poner otro punto de ruptura o, si es sólo para una vez, simplemente indicar que queremos que siga la ejecución hasta ese punto.
- Se pueden consultar las funciones que hay apiladas. Además, podemos pinchar en cualquiera de ellas de forma que el entorno muestre el punto donde se encuentra y las variables locales con sus valores.
- Podemos usar la barra de herramientas para avanzar.

Es habitual usar intensivamente las opciones de la barra de herramientas en un caso en que trazamos paso a paso. De hecho, suelen ser opciones que, teniendo un acceso rápido con teclado, se memoricen para no tener que pulsar con el ratón (las ayudas, si pasea el cursor de ratón, incluyen la tecla correspondiente). Son opciones que también están en el menú “**Debug**” pero que se repiten para un uso más rápido y cómodo. Recordemos su utilidad:

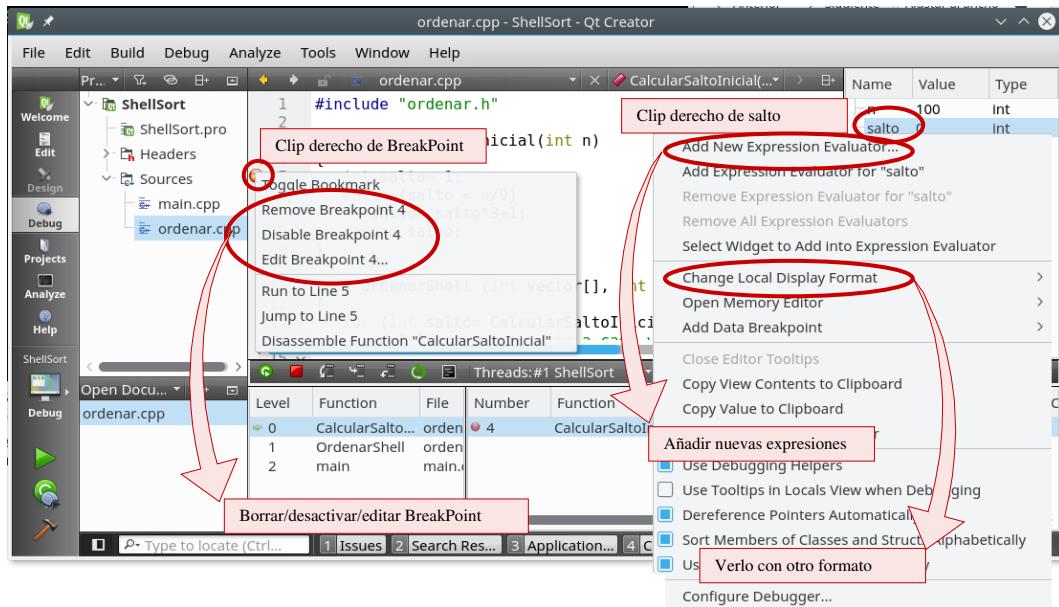
- *Continuar*. Sigue ejecutando. Por ejemplo, para dejar que alcance otro punto de ruptura o termine la ejecución para ver el resultado final.
- *Parar*. Abandonar el depurado. Por ejemplo, tras encontrar un error que implica volver al modo de edición, modificar el código y volver a empezar.
- *Avanzar sin entrar*. Cuando desea que se ejecute la línea de código actual y se avance a la siguiente línea. Se distingue de la siguiente porque...
- *Avanzar entrando*. ... en este caso entra en las llamadas de la línea actual. Si estamos en una línea que tiene llamadas a función, se puede avanzar una línea ejecutando todas esas llamadas y terminando la línea actual —opción anterior— o, como es este caso, se avanza un paso pero entrando en la llamada a función que contiene la línea actual. Lógicamente, se comporta igual si la instrucción es simple.
- *Avanzar hasta salir*. Si queremos seguir ejecutando línea a línea pero queremos que termine la ejecución de la función actual sin pararse. Por ejemplo, si entra en una función que sabe que funciona correctamente y no quiere trazarla.
- *Volver a comenzar*. Por ejemplo, si quiere volver a recorrer algún paso anterior y tiene que volver a empezar para repetir la traza.

Sí, hay otra opción más pero resulta de menor interés, al menos, para nuestros programas; es un botón para poder ver el programa que se traza a nivel de instrucciones ensamblador.

Algunas opciones recomendadas

No sólo disponemos de esos botones. Tenemos más opciones, algunas de ellas se usan de forma habitual. Como ejemplo, podemos ver las opciones que se abren cuando pinchamos con el botón derecho en un punto de ruptura —*breakpoint*— que ya está en el programa. Aparte de la opción de borrar, disponemos de la opción de desactivar —*disable*— en caso de que

no queramos perderlo pero tampoco queremos que se pare para esta ejecución. En la siguiente figura, a la izquierda, hemos presentado el menú que se abre en este caso:



Más interesante que desactivar, la opción de editar resulta en algunos casos imprescindible. Por alguna extraña razón, muchos estudiantes que empiezan la ignoran, pero hay ejecuciones en las que tenemos que editar para afinar el comportamiento del punto de ruptura. Con ello no sólo podremos especificar si está activo o no, o si sólo funcionará la primera vez que lleguemos a él, o la enésima, sino que podemos especificar con precisión la condición de parada.

Por ejemplo, suponga que tiene un programa que falla dentro de un bucle que cicla 1000 veces. La traza ha demostrado que falla cuando `i==637`. Quiere saber qué estado tiene el programa justo cuando itera con ese valor. Podemos situar un punto de ruptura al comienzo del cuerpo del bucle y usar esa condición para que se active. Note que sin esta opción, tendría que ir pulsando continuar para cada iteración. Las opciones pueden ser más complejas, incluyendo opciones compuestas que hacen que podamos parar un programa en una situación muy concreta.

Por otro lado, en la figura anterior también hemos presentado otro menú distinto, el que se abre en el caso de que pulsemos en la ventana de variables locales y expresiones. Esta ventana presenta las variables del contexto donde estamos situados, pero en muchos casos no es suficiente para entender el estado en el que nos encontramos. Si desea consultar un valor distinto, de alguna variable global, el resultado de una expresión donde aparecen las variables conocidas, puede añadir una nueva expresión a evaluar. Basta con pinchar con el botón derecho en esta ventana para añadir una subsección con nuevas consultas.

En otros entornos puede encontrar esta ventana con el nombre de «**Watches**», puesto que es una ventana donde se observa el valor de variables y expresiones de forma que en cada paso que da el depurador, se actualizan los valores para que el usuario conozca el nuevo estado sin tocar nada. Puede añadir y eliminar las expresiones que deseé para tener otros datos aparte de los que automáticamente se presentan.

Además de añadir nuevos elementos, es importante observar que dispone de una opción para cambiar el formato de presentación. En nuestro ejemplo de la figura anterior, hemos enfatizado esta opción que está activa para cambiar el formato de visualización del objeto `salto`. Observe que el menú se ha abierto pinchando en este entero. Por tanto, las opciones que nos dará para el formato están relacionadas con el tipo entero; por ejemplo, para verlo en formato hexadecimal.

El submenú que se abre cuando accedemos el formato de visualización depende del objeto que pinchamos. De especial interés serán los tipos puntero, puesto que el formato podría estar relacionado con una cadena con cierta codificación o con una vector con cierto número de elementos.

C

Git: fundamentos



Introducción	123
Control de versiones con Git	
El concepto de versión	
El repositorio local: conceptos básicos	125
Áreas de almacenamiento	
Estado de los archivos	
Configuración	
El repositorio local: diferencias.....	133
Tres áreas de almacenamiento, dos bloques de diferencias	
Mostrando diferencias	
Historial del repositorio	
Repositorios remotos.....	145
Protocolos del servidor	
El repositorio <i>origin</i>	
Avanzando con el repositorio local	
Subiendo nuestros cambios	
Descargando una nueva versión	

C.1 Introducción

El control de versiones de un proyecto software se refiere a la gestión de las distintas versiones por las que pasa cada uno de los ficheros que componen el proyecto a lo largo del tiempo. No se limita a las versiones de lanzamiento, sino al control de cada uno de los cambios que van surgiendo y que se van aplicando tanto para corregir errores como para mejorar la funcionalidad.

El interés no es tener una versión certificada de cómo va el proyecto, sino el control de cada uno de los cambios de forma que se pueda revisar el historial de cada uno de ellos. Este control permite, por ejemplo, revertir cambios para recuperar una versión anterior desde la que proseguir el desarrollo, o incluso llevar el control de varios cambios en paralelo que probablemente finalizarán con una versión resultado de la mezcla de ellos.

Si un control de versiones resulta especialmente necesario cuando un proyecto empieza a complicarse, resulta una necesidad inevitable cuando el equipo de desarrollo también aumenta. Deberá seleccionar un sistema de control de versiones para resolver este problema. Con él, podrá saber qué cambios se han dado en cada uno de los ficheros, qué versiones se han generado, quién ha cambiado qué, qué modificaciones se hicieron en paralelo, qué desarrollos se plantearon de forma experimental y no terminaron por incorporarse a la solución, etc.

El sistema de control de versiones no es más que un conjunto de herramientas software que le van a ayudar a realizar todo este proceso de una forma cómoda, resolviendo automáticamente la mayoría de estas operaciones. En concreto, vamos a presentar «[Git](#)»; nuestro objetivo será hacer una simple introducción que le permita desarrollar sus primeros programas fácilmente; una vez que no tenga dudas con lo básico, le será muy fácil estudiar esta herramienta a fondo para sacarle el máximo partido. Puede consultar la referencia Chacon y Straub[3], que cuenta con versiones libres que puede descargar, para más detalles Chacon y Straub[14].

En el caso de necesitar ayuda de una orden concreta, dispone también de la ayuda en línea de «[Git](#)». Puede escribir la línea:

```
Consola
prompt:~> git help <orden>
```

para ver un manual sobre `<orden>`. Aunque antes de eso, es conveniente que lea este breve tutorial para entender al menos los conceptos básicos que le faciliten el entender la extensa información que puede llegar a obtener en esas ayudas.

C.1.1 Control de versiones con Git

Existen múltiples soluciones para el control de versiones, desde las primeras que se crearon para controlar versiones en nuestro disco local, hasta las últimas herramientas de desarrollo distribuido. No vamos a entrar en detalles sobre la gama disponible ni su historia. En este capítulo vamos a centrarnos directamente en introducir «[Git](#)» como una solución —ampliamente usada— al problema de la gestión.

Es un sistema creado por *Linus Torvalds*, aunque cuando lo conozca verá que es una gran herramienta no sólo por su creador, sino porque es un sistema distribuido que por un lado permite que múltiples desarrolladores trabajen en paralelo como si de un proyecto local se tratase y, por otro, facilita la integración de los cambios que puedan haberse realizado por distintos equipos de forma totalmente independiente. Esto lo hace ideal para proyectos que se realizan en la comunidad de

software libre, donde el proyecto puede ser muy grande y los desarrolladores pueden llegar a ser cientos distribuidos por todo el mundo. Sin duda, el desarrollo del *kernel*¹ de *Linux* con «[Git](#)» es una garantía de que es una gran opción para la gestión de nuestros proyectos y sus futuros desarrollos.

C.1.2 El concepto de versión

Antes de comenzar, es fundamental entender qué es una versión del proyecto. Si tiene poca experiencia programando, es posible que piense que las versiones son las conocidas numeraciones que se dan a los paquetes cuando se lanzan al público. Por ejemplo, la versión *1.0* de un programa que al año siguiente se ve mejorada por la versión *1.1*. Seguramente sospechará que no es así, aunque es posible que piense que las versiones consisten en mantener ese tipo de numeraciones para cada archivo. Por ejemplo, si tiene los 5 archivos *file1.h*, *file1.cpp*, *file2.h*, *file2.cpp* y *main.cpp*, podría mantener una numeración para cada uno de estos archivos. El primero y segundo están en la versión *0.9*, los dos siguientes están ya en la *1.0*, etc. Es fácil entender que este sistema puede llegar a complicarse demasiado.

Podríamos introducir nuevas variantes para razonar sobre soluciones más prácticas y eficaces. Sin embargo —yendo al grano— nos centramos en la solución de «[Git](#)»: una versión del proyecto es una instantánea de todos los archivos que lo componen. La versión 1 tiene los 5 archivos anteriores, modificamos los dos primeros y obtenemos la versión 2; ésta, de nuevo, son los 5 archivos aunque con dos de ellos modificados. No piense en versiones asociadas a cada archivo o a cambios en una parte del proyecto. Piense que cada versión es todo el proyecto en un nuevo estado, tal vez con unos pocos cambios, con muchos cambios, o incluso con toda una parte del proyecto eliminada y otra gran cantidad de archivos añadidos.

Si entiende esta forma de gestionar el proyecto, se dará cuenta de que realmente guardar la secuencia de versiones que se van produciendo es una operación que normalmente implicará añadir nuevos pasos, dejando los anteriores como estaban. Por ejemplo, si decide borrar un archivo y generar una nueva versión del proyecto, tendrá una situación donde no existe el archivo, pero que se puede recuperar porque tiene todas las versiones en el historial. Si dice al sistema que quiere volver a la versión anterior, ese archivo que desapareció volverá a estar presente.

Como consecuencia, esta capacidad funcional del sistema hace muy simples no sólo los cambios hacia nuevas versiones, sino también aprovechar que se tienen las versiones anteriores en el historial para recuperar estados anteriores y combinar o revertir errores. No piense que el sistema guarda una y otra vez todo el proyecto, sino que gestiona el almacenamiento para ir modificando los cambios sin volver a guardar todo el contenido. En cualquier caso, desde el punto de vista del usuario, deberemos pensar en cada versión como todo el proyecto y olvidarnos de los detalles internos que permiten minimizar el espacio requerido.

Versiónes y valores hash

El sistema de «[Git](#)» usa intensivamente los valores *hash*. Una función *hash* tiene como entrada una secuencia de *bytes*, realiza una serie de operaciones con ellos, y calcula cierto número como valor *hash* correspondiente a esos *bytes*.

Es posible que haya trabajado con algún generador conocido. Por ejemplo, puede usar la orden «[md5sum](#)» con un archivo que haya bajado de internet para obtener un valor *hash* del archivo. Este valor es un número binario de 128 bits, es decir, un valor de 32 dígitos hexadecimales. Se usa para comprobar la integridad de los archivos. La idea es que baja el archivo y le informan del correspondiente valor *md5* para que pueda comprobar que su archivo es el original.

Lógicamente, podemos proponer múltiples algoritmos y longitudes *hash*. Lo ideal es que si un archivo cambia, el valor *hash* también lo haga. No se puede garantizar que dos archivos distintos no tengan el mismo valor *hash* pero los algoritmos se han diseñado para que sea muy poco probable. En caso de necesitar menos coincidencias, la solución está en aumentar la longitud del valor *hash*. Por ello, es posible que encuentre varias órdenes en su sistema para calcular distintos valores *hash*, como «[sha256sum](#)» o «[sha512sum](#)».

El sistema de «[Git](#)» se basa en el algoritmo *SHA-1*. Este algoritmo genera una secuencia de *160 bits*, es decir, 40 dígitos hexadecimales. Pruebe a ejecutar en su máquina el programa «[shasum](#)» sobre algún archivo; le escribirá en la salida estándar la correspondiente secuencia. Por ejemplo, si calcula el valor que corresponde al archivo que ahora mismo estoy editando, el resultado es:

```
prompt: ~> shasum git.tex
36f292d0ccc01b85cf6f49ff39c7448ea41ef17c git.tex
```

ahora que he introducido estas líneas, volver a ejecutarlo da como resultado:

```
prompt: ~> shasum git.tex
e720c88fe2ada59025c123968ceb6c0f77c228e0 git.tex
```

donde puede observar que un pequeño cambio ha provocado que el número sea radicalmente distinto.

¹Un proyecto que incluye millones de líneas de código y en que han contribuido miles de programadores.

Para resolver el problema de las versiones, «Git» “mata dos pájaros de un tiro:” usa un valor *SHA-1* para archivar la nueva versión —esto permite un control sobre la integridad de los contenidos— y por otro lado usa la secuencia de 40 dígitos hexadecimales resultante como un identificador de la versión.

Por tanto, olvídense de versiones 1.0, 1.1, etc. Eso se usará cuando tenga que lanzar una versión del proyecto para el usuario final y se resolverá de una forma muy determinada con «Git». Ahora, lo que tiene que recordar es que cada versión es una cadena de 40 caracteres hexadecimales. No se preocupe, no tendrá que ir copiando largas cadenas en sus órdenes, para eso siempre podrá usar copiar/pegar, alguna herramienta que le solucione el problema o, mucho más sencillo, usar sólo los primeros caracteres de la cadena. Tenga en cuenta que aunque sean muy pequeños los cambios, la cadena *SHA-1* cambiará completamente.

Como veremos más adelante, «Git» facilita los listados y las referencias indicando solamente una pequeña parte: normalmente los primeros 7 caracteres. Con esta simplificación, podemos decir que la primera versión del fichero `git.tex` tenía asociado el número «`36f292d`» y en la segunda teníamos el número «`e720c88`».

C.2 El repositorio local: conceptos básicos

La mayor parte del trabajo con «Git» se realiza de forma local, como si no hubiera otros desarrolladores, ni tampoco un servidor central con el proyecto común compartido. En esta sección presentamos los detalles de este flujo de trabajo; más adelante veremos las nuevas posibilidades cuando interactuamos con otros repositorios.

Un repositorio es, como su nombre indica, una almacén donde se guardan cosas. En nuestro caso, es el lugar donde se almacenarán todas y cada una de las versiones de nuestro proyecto. Podemos pensar que es algo que acumula nuevas versiones. Tenga en cuenta que si añadimos un nuevo archivo al proyecto podemos generar una nueva versión. Si luego lo quitamos y generamos una nueva versión, generamos una tercera, no volvemos a la primera. Recordemos que es una secuencia de nuevas instantáneas del estado del proyecto.

Vamos a desarrollar la lección sobre un ejemplo extremadamente simple donde hacemos un programa que tiene algunas funciones que incluyen un simple mensaje. Forzaremos la división en archivos para poder ver cómo evoluciona en proyecto con «Git».

Para poder comenzar, configuraremos la información de qué usuario somos. Tenga en cuenta que el sistema de gestión de versiones controlará la evolución del proyecto anotando quién hace qué. Por tanto, primero debemos establecer nuestra identidad. Para ello, lanzamos las órdenes:



```
prompt:~> git config --global user.name "Antonio Garrido"
prompt:~> git config --global user.email "antonio@buzon.ugr.es"
```

que modifican un archivo global de configuración para «Git». Sí, si hay una configuración global es porque hay una “local”². Más adelante verá que podemos establecer estos parámetros para que afecten solamente a un proyecto concreto, por ejemplo, si tiene un proyecto en su empresa pondrá su identidad profesional mientras que si está en un proyecto de software libre tal vez le interese una identidad independiente.

C.2.1 Áreas de almacenamiento

Por ahora vamos a trabajar con un repositorio local, aunque luego veremos que en la práctica se realizan las mismas acciones cuando trabajamos con nuestra copia del proyecto. Eso significa que, para empezar, en nuestro disco vamos a tener dos zonas de almacenamiento:

- Nuestro *directorio de trabajo*. Lo conoce de manera sobrada, es lo que ha estado siempre usando. Tiene un proyecto en un directorio —tal vez con subdirectorios— que contienen distintos archivos. Son los archivos que editamos y que componen el fuente del proyecto. En nuestro caso, estamos trabajando en el directorio `mensajes`.
- El *repositorio local*. Cuando hacemos una modificación y queremos almacenar una nueva versión de nuestro proyecto, le tenemos que decir a «Git» que lo guarde en el repositorio local. «Git» se encarga de gestionar una zona de almacenamiento donde va acumulando las distintas versiones.

Para empezar, creamos un directorio `mensajes` para el proyecto donde añadimos un único archivo `saludar.cpp` que contiene un mensaje de inicio del programa. El directorio `mensajes` es el directorio de trabajo. Cuando el proyecto crezca, podrá incluir nuevos archivos y subdirectorios. El estado inicial es:



```
prompt:~> .
| -- mensajes
`-- saludar.cpp
```

²Realmente hay más, uno de sistema (opción `-system`) otro de usuario (opción `-global`) y otro para un proyecto (sin opción).

donde hemos añadido el archivo `saludar.cpp` con el siguiente contenido:

```
#include <iostream>
using namespace std;
int main() {
    cout << "Comenzamos la sesión..." << endl;
}
```

Sólo tenemos el directorio de trabajo. Si queremos que este proyecto se gestione con «Git», será necesario indicarle que prepare el área de repositorio local. Para ello, no hay más que usar la orden `init` de la siguiente forma:

```
prompt:~/mensajes> git init
Initialized empty Git repository in /home/antonio/mensajes/.git/
```

Observe que lo hemos ejecutado en el directorio raíz de nuestro proyecto. Aunque hay otras opciones, es lo más simple, lanzarlo en ese directorio donde «Git» sabe que debe crear el área de repositorio. Como consecuencia, crea un directorio “oculto” para guardar todo lo necesario. Si exploramos los contenidos de nuestro directorio de trabajo descubrimos que se han añadido unos cuantos ficheros y directorios:

```
.
|-- .git
|   |-- branches
|   |-- config
|   |-- description
|   |-- HEAD
|   |-- hooks
|   |-- info
|   |-- objects
|   '-- refs
`-- saludar.cpp
```

donde hemos eliminado algunos detalles en subdirectorios por simplicidad.

No se preocupe demasiado por estos nombres. El directorio `.git` está oculto porque el usuario normalmente no tiene que tenerlo en cuenta. Es algo que gestiona «Git» y que nosotros ignoramos. Lo que debe tener en cuenta es que ahora nuestro proyecto está preparado para empezar a archivar versiones gestionadas por «Git».

Área de preparación

No lo hemos contado todo; realmente hay una tercera área o zona de almacenamiento. En principio es muy razonable hablar de las dos que hemos presentado —directorio de trabajo y repositorio— pero en la práctica verá que es conveniente gestionar los cambios y nuevas versiones añadiendo una más entre estas dos que hemos presentado.

La idea es que cuando creamos una nueva versión es probable que cambiemos varios archivos. Por ejemplo, imagine que va a añadir dos nuevos archivos `generar.h` y `generar.cpp`. Podría editar el primero y añadirlo al repositorio, sin embargo, parece poco razonable que generemos una nueva versión con el archivo `generar.h` pero sin el archivo `generar.cpp` que completa el cambio. Al escribir el primero, ya lo tenemos preparado para que forme parte de la siguiente versión, aunque seguimos editando el segundo para completar los cambios. Una vez que hemos terminado los dos, podemos indicar a «Git» que añada una nueva versión al repositorio.

Para gestionar esta forma de trabajo se usa el *área de preparación* (del inglés *staging area*, aunque también conocida como *index*). El resultado es que disponemos de tres zonas independientes:

- El *directorio de trabajo*.
- El *área de preparación*.
- El *repositorio local*.

El hecho de que esta nueva zona se llame también índice —del inglés *index*— no debería confundirle. No significa que sea un “apunte” sobre qué ficheros están o no preparados. Es realmente un área de almacenamiento; si mandamos un archivo a esta zona, el archivo se guarda con todo su contenido. Realmente, la orden de inicialización del repositorio local ha generado una situación como la que se presenta en la figura C.1, con tres áreas, aunque las dos últimas vacías.

Para generar una nueva versión realizamos dos pasos:

1. *Añadimos los archivos* que formarán parte de los cambios en el área de preparación. Esto se puede hacer con la orden `add`.
2. *Confirmamos los cambios* creando una nueva versión, es decir, «Git» procesa los cambios que hay archivados en el área de preparación para pasarlos al repositorio local como nueva versión del proyecto. Esto se puede hacer con la orden `commit`.

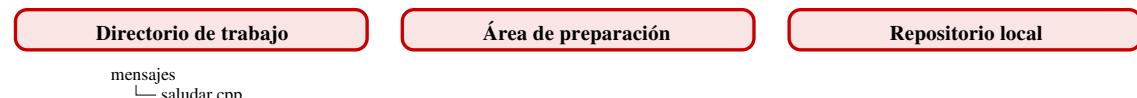


Figura C.1
Zonas de almacenamiento.

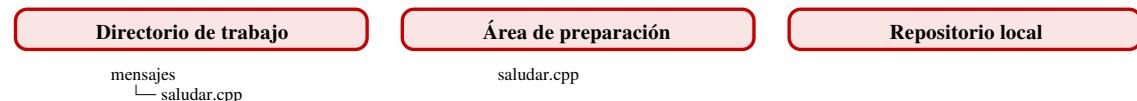
En nuestro caso, añadimos el archivo `saludar.cpp` al área de almacenamiento:

```

Consola
prompt:~/mensajes> git add saludar.cpp

```

que provoca el siguiente resultado:



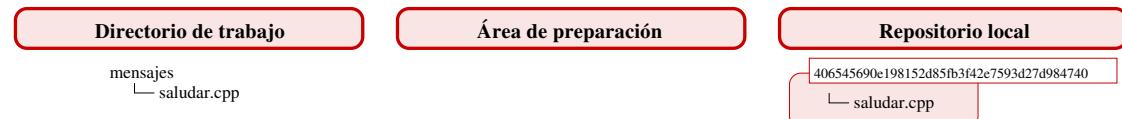
a partir del que podemos crear una nueva versión con la siguiente orden:

```

Consola
prompt:~/mensajes> git commit -m "Versión inicial"
[master (root-commit) 4065456] Versión inicial
 1 file changed, 5 insertions(+)
 create mode 100644 saludar.cpp

```

Observe que hemos añadido una opción `<-m>` con un mensaje entre comillas. Esta cadena es un comentario que permite documentar la versión que estamos generando. Esta orden implica una nueva situación de nuestras zonas de almacenamiento:



Preguntemos a «Git» en qué estado estamos:

```

Consola
prompt:~/mensajes> git status
En la rama master
nothing to commit, working directory clean

```

No tenemos nada en el área de preparación ni nada nuevo en el directorio de trabajo. Pero... ¿rama *master*? Más adelante hablaremos de ramas. Ahora mismo, asuma que la rama es la secuencia de versiones que se van encadenando en el repositorio local. El hecho de llamarla *master* no es nada especial, de hecho, podríamos considerar otros nombres aunque lo normal es usar éste, el que por defecto asigna «Git» a la “rama principal”.

C.2.2 Estado de los archivos

Sólo con los primeros pasos ya se habrá dado cuenta de que podemos tener muchas versiones de un mismo archivo. Realmente todas las que queramos, pues cuando modificamos un archivo y lo guardamos en una nueva versión no perdemos el antiguo, pues queda en el historial del repositorio. Pero además, es posible tener un archivo en el directorio de trabajo e incluso en el área de preparación. ¿Qué posibles estados puede tener un archivo? Para resolverlo, ampliamos nuestro proyecto.

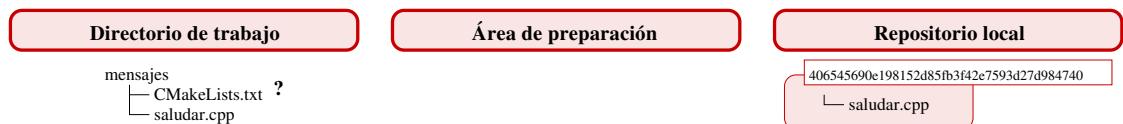
Para empezar, vamos a añadir un fichero `CMakeLists.txt` para gestionar nuestro proyecto con «`CMake`». Editamos este archivo con el siguiente contenido:

```

1 project ( MENSAGES )
2 cmake_minimum_required (VERSION 3.5)
3 set ( MENSAGES_SOURCES saludar.cpp )
4 add_executable ( saludar ${MENSAGES_SOURCES} )

```

En este archivo simplemente indicamos que el proyecto consiste en el ejecutable `saludar` que se obtiene compilando esos fuentes (ahora mismo, un solo archivo). Esta cambio implica una nueva situación en nuestro proyecto:



Lo que no afecta a los contenidos que «Git» está gestionando, pues no tiene constancia de este archivo ni en el área de preparación ni en el repositorio. Sin embargo, es un archivo del área de trabajo. Podemos preguntar a «Git» en qué estado estamos:

```
Consola

prompt:~/mensajes> git status
En la rama master
Archivos sin seguimiento:
(use «git add <archivo>...» para incluir en lo que se ha de confirmar)

CMakeLists.txt

no se ha agregado nada al commit pero existen archivos sin seguimiento (use «git add»
para darle seguimiento)
```

El estado del archivo `CMakeLists.txt` es *sin seguimiento* (del inglés *untracked*). Nunca se ha insertado en el área de preparación y menos en el repositorio, por tanto, «Git» no lo está siguiendo.

Observe que el estado nos ofrece algún mensaje de ayuda para recordarnos las opciones que podrían interesarnos. En este caso, sugiere que añadamos el archivo al área de preparación. Podemos hacerlo como sigue:

```
Consola

prompt:~/mensajes> git add CMakeLists.txt
```

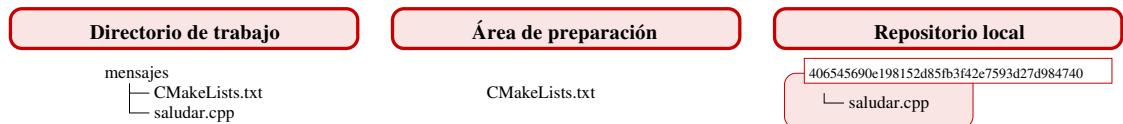
Si hacemos una consulta de estado obtenemos:

```
Consola

prompt:~/mensajes> git status
En la rama master
Cambios para hacer commit:
(use «git reset HEAD <archivo>...» para sacar del stage)

nuevo archivo: CMakeLists.txt
```

donde se nos informa que ese archivo es nuevo para el siguiente *commit*. Además, nos ayuda informando de cómo podríamos eliminarlo del área de preparación. La situación actual es la siguiente:

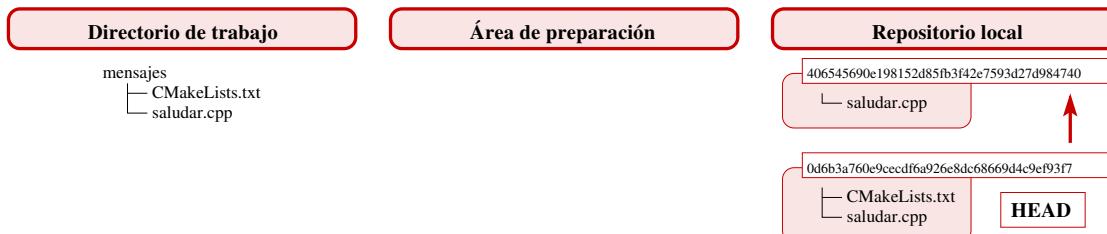


En este caso el archivo ha dejado de estar *untracked* puesto que ahora está *preparado* (del inglés *staged*). Podemos confirmar el cambio actualizando el repositorio local con los cambios que están preparados:

```
Consola

prompt:~/mensajes> git commit -m "Configuración cmake"
[master 0d6b3a7] Configuración cmake
 1 file changed, 5 insertions(+)
 create mode 100644 CMakeLists.txt
```

que nos lleva a la siguiente situación:



En este punto es interesante que se fije en los detalles:

- En el mensaje que nos devuelve la última operación *commit*, se nos ha indicado que estamos en la rama *master* seguida de un pequeño número. Este número corresponde a los 7 primeros dígitos del *SHA-1* que identifica la versión.
- En la figura hemos añadido una flecha para enfatizar el hecho de que «**Git**» sabe que esa versión se deriva de la anterior. Más concretamente, decimos que la versión «**4065456**» es padre de «**0d6b3a7**».
- Hemos añadido una nueva etiqueta «**HEAD**» en la versión actual del proyecto. El repositorio puede tener múltiples instantáneas, pero nosotros siempre estaremos situados en una de ellas. Cuando se confirman nuevos cambios, la nueva instantánea —que será “hija” de la actual— pasará a ser la nueva «**HEAD**».

En esta nueva situación los dos archivos están siendo seguidos (del inglés *tracked*). Su estado ya no es *staged*, puesto que están confirmados. Ahora están *sin modificar*.

Modificar-preparar-confirmar

En el siguiente paso vamos a modificar los dos archivos; en concreto, hacemos algún cambio para mejorar el formato introduciendo algunos saltos de línea y espacios. Por ejemplo, el archivo **CMakeLists.txt** ahora lo formateamos así:

```

1 project ( MENSAJES )
2 cmake_minimum_required (VERSION 3.5)
3
4 set ( MENSAJES_SOURCES
5       saludar.cpp
6     )
7
8 add_executable ( mensaje ${MENSAJES_SOURCES} )

```

Una vez que hemos editado y salvado a disco los dos archivos, podemos preguntar por el estado del proyecto:

```

Consola

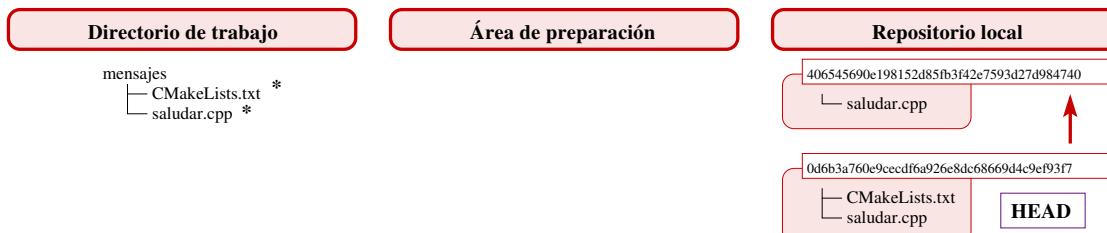
prompt:~/mensajes> git status
En la rama master
Cambios no preparados para el commit:
  (use «git add <archivo>...» para actualizar lo que se confirmará)
  (use «git checkout -- <archivo>...» para descartar cambios en el directorio de trabajo)

    modificado:   CMakeLists.txt
    modificado:   saludar.cpp

no hay cambios agregados al commit (use «git add» o «git commit -a»)

```

En este momento, los dos archivos están en estado *modificado*. Lo demás está exactamente en las mismas condiciones. El esquema puede ser el siguiente:



Si seguimos las indicaciones que hemos visto, ahora tendríamos que ordenar **add** para cada uno de los archivos para prepararlos de forma que el siguiente **commit** los confirmara. Sin embargo, lo vamos a realizar de una forma más rápida: indicando que se preparen todos los archivos cambiados. La orden es:

Consola

```
prompt: ~/mensajes> git add .
```

donde hemos indicado el directorio actual en lugar de un archivo concreto. El efecto es que todos los archivos de este directorio y sus subdirectorios, que hayan sido cambiados y estén siendo seguidos, se añaden a la zona de preparación. Por tanto, el nuevo estado es:

Consola

```
prompt: ~/mensajes> git status
En la rama master
Cambios para hacer commit:
(use «git reset HEAD <archivo>...» para sacar del stage)

  modificado:    CMakeLists.txt
  modificado:    saladar.cpp
```

lo que corresponde al siguiente esquema:

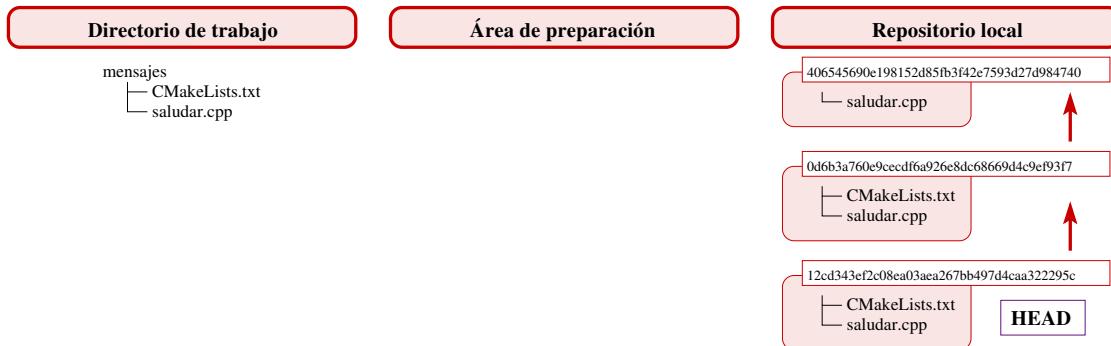


Podemos salvar los cambios preparados con la siguiente orden:

Consola

```
prompt: ~/mensajes> git commit -m "Formateo"
[master 12cd343] Formateo
 2 files changed, 9 insertions(+), 4 deletions(-)
```

lo que da lugar a una nueva situación que podemos representar así:



Note que la secuencia de modificación de archivos, preparación y confirmación será una operación especialmente habitual cuando avancemos en el desarrollo del proyecto. Para simplificar la gestión, «**Git**» permite realizar una operación que resuelva todo a la vez. En lugar de hacer `«add .»` y `«commit»`, podríamos haber obtenido el mismo resultado con la siguiente orden:

Consola

```
prompt: ~/mensajes> git commit -a -m "Formateo"
[master 12cd343] Formateo
 2 files changed, 9 insertions(+), 4 deletions(-)
```

Podríamos decir que pasa los archivos modificados que están siendo seguidos directamente al repositorio local para obtener la siguiente versión. Observe que deben ser archivos que *están siendo seguidos*, es decir, un archivo *untracked* no va a añadirse con esta orden, debe añadirlo inicialmente de forma explícita.

Eliminando y renombrando

El proceso para borrar un archivo es similar a lo que se ha hecho para modificar, aunque ahora no se guarda una versión del archivo en la zona de preparación, sino una referencia a que se ha borrado el archivo. Como cuando hacemos cambios, se puede hacer en tres etapas:

1. Borramos el archivo de nuestro directorio de trabajo. Por ejemplo, con la orden `rm` de nuestro terminal.
2. Indicamos que el cambio pase al área de preparación.
3. Confirmamos el borrado con la orden `commit`.

aunque lo más sencillo es realizar las dos primeras en un único paso, con el borrado de «`Git`»:

```
Consola
prompt: ~/mensajes> git rm <archivo>
```

que además de borrar el archivo lo deja preparado para el siguiente `commit`.

De forma similar, puede usar la orden para renombrar un archivo:

```
Consola
prompt: ~/mensajes> git mv <antiguo> <nuevo>
```

que hace las dos operaciones: renombrar el archivo y preparar el cambio en la zona de preparación. En este caso, es interesante saber que «`Git`» nos va a mostrar la ayuda relacionada con renombrar, aunque realmente gestiona un cambio de nombre como un cambio en el directorio de trabajo más un *borrado/añadido* en el repositorio.

C.2.3 Configuración

Una vez que ha entendido el formato básico y la dinámica de un repositorio local, es más fácil incluir algunas indicaciones más sobre la configuración que le serán útiles para completar lo que ha aprendido y para las siguientes secciones de este tutorial.

No es intención realizar una exposición extensa, sino sólo añadir las opciones más básicas para que le resulte fácil su comienzo con «`Git`». Puede ejecutar en su sistema:

```
Consola
prompt: ~> git help config
```

para comprobar la inmensa cantidad de opciones que puede usar. En principio, con establecer unas pocas opciones puede realizar prácticamente cualquier desarrollo. En esta sección, proponemos unas cuantas básicas que probablemente le sean útiles.

Identidad para un proyecto

Las secciones anteriores se han desarrollado con la identidad establecida en nuestra configuración global. Al comienzo de la sección C.2 indicamos cómo podíamos hacerlo incluyendo la opción `--global`, lo que implicaba un cambio en un archivo de usuario —por ejemplo, `~/.gitignore` de nuestro `home`— que nos servía para todos los proyectos.

Si no incluimos esta opción `--global`, estamos indicando que la opción de configuración se establece para el proyecto actual. Si ejecutamos la orden de configurar en el directorio `mensajes`:

```
Consola
prompt: ~/mensajes> git config user.name "Fenix"
prompt: ~/mensajes> git config user.email "fenix@developers.com"
```

cambiará la identidad para este proyecto —cambia el archivo `.git/config`— de forma que las nuevas entradas en el historial se marcarán con estos datos.

Ignorando archivos

Las tareas de desarrollo no se limitan a añadir los archivos de código del programa. También existen otros ficheros relacionados que no contienen código y que es posible que no queramos incluir en el repositorio. Si «`Git`» encuentra nuevos archivos o subdirectorios que cuelgan de nuestro directorio de trabajo, va a considerarlos para su posible gestión automática.

Por ejemplo, suponga que para ejecutar el proyecto anterior he usado en entorno de desarrollo «`qtcreator`» que puede manejar el proyecto con la configuración de «`CMake`». Este *IDE* genera automáticamente un archivo `CMakeLists.txt.user` en el directorio de trabajo. La próxima vez que pregunte a «`Git`» sobre su estado le informará sobre este nuevo archivo.

Probablemente no le interese que el repositorio lo contenga; además, también querrá evitar añadirlo accidentalmente cuando incluye archivos múltiples. Lo más sencillo es configurar que se ignore.

Para resolver el problema, podemos generar un nuevo archivo en el directorio raíz indicando lo que ha de ignorarse³. El nombre es `.gitignore` y contiene texto indicando los archivos y directorios que deberá ignorar. En nuestro caso, podemos usar:

```
# Ficheros relacionados con el entorno qtcreator
CMakeLists.txt.user
```

donde podrá notar que hemos añadido una línea que comienza con el carácter `'#'` para escribir un comentario que no afecta al resultado. En este fichero podemos añadir las líneas que queramos, indicando nombres de archivos, directorio o incluso usando caracteres comodín para indicar múltiples posibilidades. Por ejemplo, el siguiente podría ser una buena opción para sus primeros proyectos:

```
# Las versiones antiguas de editores
*~
*.bak

# Ficheros swp de vi
*.swp

# Directorios de compilación que puedan generarse aquí
BUILD*
build*

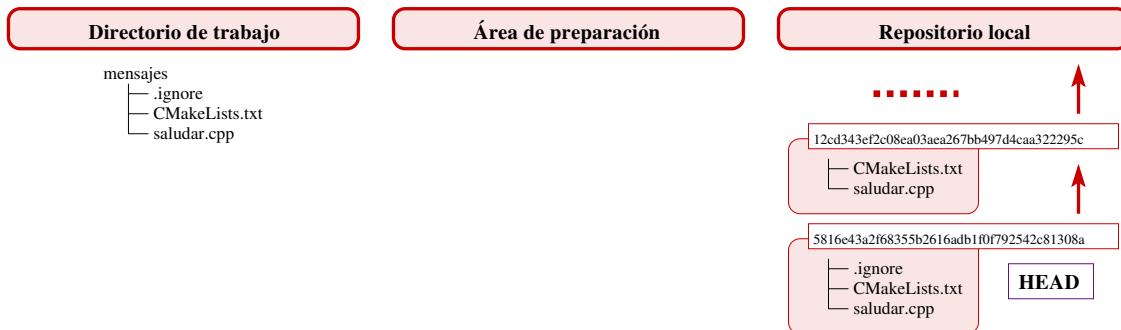
# Ficheros relacionados con el entorno qtcreator
CMakeLists.txt.user*
```

En este ejemplo evitamos algunos nombres que podrían generarse de forma inesperada. Además, también directorios que comienzan con `<build>`, por si desea generar ejecutables en el directorio de trabajo.

Como archivo que forma parte del proyecto, podemos añadirlo al repositorio. Es un archivo nuevo, está sin seguimiento, lo añadimos a la zona de preparación y confirmamos el cambio:

```
prompt:~/mensajes> git add .gitignore
prompt:~/mensajes> git commit -m "Ficheros a ignorar"
[master 5816e43] Ficheros a ignorar
 1 file changed, 13 insertions(+)
 create mode 100644 .gitignore
```

El resultado es una nueva versión donde ya tenemos tres archivos:



Por último, es interesante indicar que podemos usar el carácter `'!'` delante de una de las líneas del archivo `.gitignore` para indicar `«no ignorar»`. Por ejemplo, podemos ignorar todos los archivos terminados en `“txt”` y luego añadir una línea para tener en cuenta cierto archivo que sí deberá gestionar.

Editando mensajes de confirmación

Un detalle que se ha evitado para no oscurecer más la discusión se refiere a la edición de mensajes de confirmación. En los ejemplos anteriores siempre hemos usado la opción `«-m»` para indicar el mensaje que acompaña a la confirmación de un `commit`. Sin embargo, esa es la versión simplificada para un cambio que sólo requiere una frase corta para su descripción.

³También se podría configurar en un archivo `exclude` oculto en el directorio de `«Git»`. En principio, resolvemos con esta opción que nos permite hacer un seguimiento del archivo como parte del proyecto.

Cuando un proyecto es más complejo, es probable que un cambio requiera mucho más que una simple frase para hacer una descripción precisa. Para ello, lo único que tenemos que hacer es un **commit** sin incluir el mensaje. En este caso, la orden **git** entiende que debe lanzar un editor de texto para que escriba los comentarios. Lanzará el que tenga configurado por defecto a menos que especifique uno concreto. Para cambiarlo, puede usar una opción de configuración. Lo más recomendable es que lo haga de forma global, para que todos sus proyectos la compartan. Por ejemplo, en mi caso he configurado el editor **kwrite** con la siguiente orden:

```
prompt: ~> git config --global core.editor kwrite
```

Con esta configuración, cada vez que haga un **commit** sin indicarle ningún mensaje, se ejecutará esta orden para editarla. Cuando termine de editarla, sálvelo y salga del editor. En caso de que no salve nada —el mensaje sería vacío— la orden de confirmación no se llevaría a cabo.

Otros parámetros configurables

Existen más posibilidades de configuración que le resultarán especialmente interesantes. Retrasaremos su presentación hasta que puedan justificarse de forma práctica. Ejemplos son los **alias** (se incluirá algún ejemplo en C.3.3, página 142) o la configuración de la mezcla (que veremos en C.4.5, página 152).

C.3 El repositorio local: diferencias

Los contenidos de la sección anterior completan una gama de posibilidades suficiente para resolver la mayoría de las operaciones que tendrá que realizar en su repositorio local. En gran medida, incluirá nuevos archivos, modificará otros, eliminará o renombrará, para finalmente archivar nuevas versiones tras los cambios. Sin embargo, es interesante mostrar una nueva perspectiva sobre el control de cambios que realiza «Git» de forma que sea más fácil entender los efectos de nuestras acciones y los mensajes que se muestran.

Es probable que con lo que hemos visto, ya haya empezado a “anotar” órdenes a modo de recordatorio de cómo hacer qué. Además, estará agradecido de ver esos mensajes automáticos en los que se nos sugiere cómo realizar la siguiente acción. Realmente puede ser muy útil, pero resulta conveniente entender qué está pasando porque si alguna vez realiza algo sin querer o da un paso que no estaba en ese recetario, podría encontrarse con una nueva situación en la que no entiende bien en qué estado se encuentra y cómo resolverlo.

C.3.1 Tres áreas de almacenamiento, dos bloques de diferencias

La primera versión que hemos explicado sobre el directorio de trabajo, el área de preparación o el repositorio hacen referencia a *áreas de almacenamiento*. Como hemos visto, es ideal para empezar a trabajar y para comprender las órdenes más habituales. En la mayoría de las referencias encontrará este enfoque para introducir «Git».

Sin embargo, en esta sección vamos a enfatizar cómo esas tres zonas se caracterizan no por lo que tienen, sino por sus diferencias. Por un lado la diferencia entre el directorio de trabajo y el área de preparación y por otro entre ésta y el repositorio. Después de ello, seguramente entenderá mejor los mensajes de estado del proyecto y muchas de las órdenes que puede usar.

Es interesante enfatizar la importancia de gestionar una diferencia en lugar de almacenar dos versiones de un mismo archivo. Aunque interpretamos la secuencia de versiones en el repositorio como una secuencia de múltiples versiones de todo el proyecto, es lógico que la gestión que realiza «Git» se haga más como una secuencia de diferencias en lugar de un almacenamiento repetitivo. No tiene sentido que si varias versiones tienen exactamente el mismo archivo, éste se almacene varias veces⁴.

Esta idea de gestión de diferencias está presente también cuando consideramos el estado del directorio y del área de preparación. Para mostrar este punto de vista, vamos a realizar una serie de operaciones simples sobre nuestro repositorio de ejemplo. Con ellas, veremos que el estado y los efectos de nuestras órdenes reflejan las diferencias que hay entre las distintas áreas.

En cualquier caso, no debería crearle confusión. Las sucesivas versiones del repositorio consideran todo el proyecto. Desde un punto de vista conceptual, cuando hablamos de una versión estamos hablando de una instantánea que incluye todos los archivos, aunque algunos de ellos no hayan cambiado. El hecho de enfatizar las diferencias no deja de ser una forma natural de entender los cambios, no una forma derivada del almacenamiento o gestión de las versiones. Tenga en cuenta que si le dice a un colega que ha generado una nueva versión del proyecto, seguramente la respuesta que recibirá será: ¿qué ha cambiado?

Directorio de trabajo vs “siguiente versión” vs HEAD

Comenzamos modificando un único archivo —**saluda.cpp**— cambiando un par de detalles para poder registrar un archivo modificado:

⁴Internamente, si un archivo es el mismo, sólo será necesario un enlace al anterior. Si un archivo cambia, podemos considerar que se almacenarán las dos versiones; a pesar de eso, es posible que se almacene uno de ellos y la diferencia con el otro, aunque será algo interno de «Git» para optimización de recursos.

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

int main() {
    cout << "Comenzamos la sesión." << endl;
}
```

Comprobamos el estado:

```
prompt:~/mensajes> git status
En la rama master
Cambios no preparados para el commit:
  (use «git add <archivo>...» para actualizar lo que se confirmará)
  (use «git checkout -- <archivo>...» para descartar cambios en el directorio de trabajo)

    modificado:      saludar.cpp

no hay cambios agregados al commit (use «git add» o «git commit -a»)
```

donde vemos que hay un archivo modificado en el directorio de trabajo, pero no hay nada en el área de preparación para el siguiente **commit**.

Añadimos el archivo al área de preparación y comprobamos el estado:

```
prompt:~/mensajes> git add .
prompt:~/mensajes> git status
En la rama master
Cambios para hacer commit:
  (use «git reset HEAD <archivo>...» para sacar del stage)

    modificado:      saludar.cpp
```

Ahora no tenemos ningún archivo modificado en el área de trabajo, aunque sí que tenemos una operación preparada para hacer el «**commit**». Volvemos a modificar el archivo **saludar.cpp** deshaciendo la edición anterior:

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

int main()
{
    cout << "Comenzamos la sesión..." << endl;
}
```

y comprobamos el estado de nuestro repositorio:

```
prompt:~/mensajes> git status
En la rama master
Cambios para hacer commit:
  (use «git reset HEAD <archivo>...» para sacar del stage)

    modificado:      saludar.cpp

Cambios no preparados para el commit:
  (use «git add <archivo>...» para actualizar lo que se confirmará)
  (use «git checkout -- <archivo>...» para descartar cambios en el directorio de trabajo)

    modificado:      saludar.cpp
```

En este momento tenemos dos bloques diferenciados en el estado de nuestro repositorio: el primero se refiere a que tenemos un archivo que podríamos confirmar para obtener una nueva versión y el segundo a que tenemos un archivo que podríamos añadir al área de preparación. Podemos verlo como dos bloques de diferencias:

- Diferencia entre el área de preparación y la última versión, «**HEAD**».
- Diferencia entre el directorio de trabajo y el área de preparación.

Donde estamos considerando el área de trabajo no como un almacén de algunos archivos, sino como la “*siguiente versión*”. Resulta especialmente interesante pensar que tenemos tres versiones de nuestro proyecto:

1. Versión confirmada: «HEAD».
2. Siguiente versión (por confirmar): área de preparación.
3. Futura versión, en edición: área de trabajo.

Cuando comprobamos el estado de nuestro repositorio la orden «git status» se encarga de comparar las diferencias que hay entre esas tres versiones e informar de ellas junto con alguna recomendación sobre cómo podemos actuar. En la situación actual, las diferencias están en el archivo `saludar.cpp`:



donde hemos indicado —con «v1» y «v2»— que las versiones del archivo de «HEAD» y del directorio de trabajo son la misma.

Como indica el estado que acabamos de consultar, la orden «`reset HEAD archivo`» nos permite “sacar del stage” (la orden `reset` tiene varias posibilidades —luego la revisitaremos— aunque ahora nos interesa el formato que nos sugiere `status`). Concretamente, lo que hace la instrucción es coger el archivo de «HEAD» y “copiarlo” a la zona de preparación. Es decir, elimina la diferencia entre estas dos zonas rectificando el archivo del “*stage*”. La operación la podemos representar así:



Pero si observa detenidamente la figura comprobará que no hemos introducido nada en la zona de preparación, sino que hemos hecho que el archivo que será parte del siguiente `commit` coincida, o lo que es lo mismo, no es un archivo que consideramos preparado. De alguna forma, “lo hemos sacado”. Pero observe también el directorio de trabajo. No hemos tocado nada en esta parte, pero ¡el archivo es el mismo! Efectivamente, si preguntamos sobre el estado:

```
Consola
prompt: ~/mensajes> git status
En la rama master
nothing to commit, working directory clean
```

Realmente podemos representar la orden como una forma de eliminar el archivo del área de preparación, como indica explícitamente la orden `status` cuando ha sugerido que podemos “sacar del stage”:



aunque ahora —espero— le haya quedado más claro por qué esa orden que parecía destinada a cambiar la zona de preparación a afectado también al estado de su directorio de trabajo.

Borrar en el repositorio pero mantener el archivo

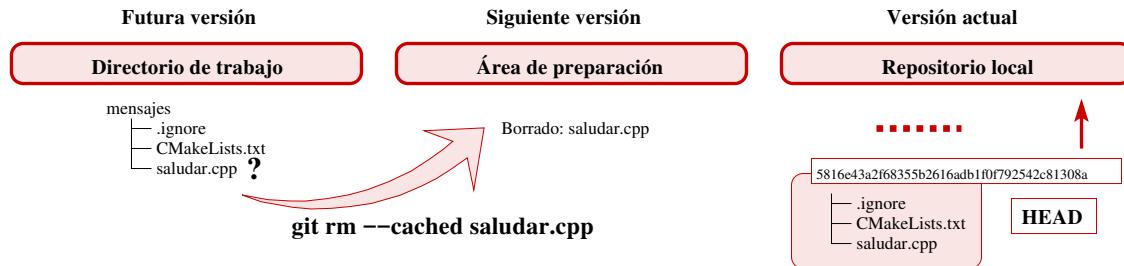
Otro ejemplo para comprobar este punto de vista de diferenciación entre áreas nos permite presentar una orden para borrar un archivo sin eliminarlo del área de trabajo.

Recuerde que en las secciones anteriores se presentó la orden `git rm` para eliminar un archivo del directorio de trabajo y dejar en el área de preparación su borrado. En alguna ocasión podría desear eliminar un archivo del repositorio pero quiere mantenerlo en el directorio de trabajo. Por ejemplo, si se equivocó al añadirlo porque no estaba en el `ignore` y desea quitarlo, pero no quiere perder el original.

En este caso, sólo queremos realizar el borrado en el repositorio, es decir, sólo dejar en el área de preparación el borrado. Por ejemplo, suponga que queremos eliminar el archivo `saludar.cpp`. Esto se puede hacer con la orden:

```
prompt: ~/mensajes> git rm --cached saludar.cpp
```

Realmente no lo borra del «HEAD», sino que prepara en la siguiente versión la eliminación del archivo. De alguna forma, podemos considerar que el efecto es el siguiente:



Antes de continuar leyendo, piense un momento en la figura anterior y deduzca el estado del repositorio... ¿Lo tiene?

- Por un lado, tenemos una diferencia entre el «HEAD» y el `stage`, indicando que hay que hacer `commit`. ¿Cómo se podría eliminar esa diferencia?
- Por otro, hay una diferencia entre la versión del `stage`, lo que será la siguiente versión del repositorio y lo que tenemos en nuestro directorio de trabajo. ¿Cómo cree que aparecerá en el estado?

Si más dilación, si pregunta por el estado del repositorio obtendrá lo siguiente:

```
prompt: ~/mensajes> git status
En la rama master
Cambios para hacer commit:
  (use «git reset HEAD <archivo>...» para sacar del stage)

      borrado:      saludar.cpp

Archivos sin seguimiento:
  (use «git add <archivo>...» para incluir en lo que se ha de confirmar)

      saludar.cpp
```

que responde exactamente a las diferencias: «HEAD» tiene un archivo que no tiene el `stage`, por tanto, sería necesario un `commit` para confirmar o deshacer el cambio eliminando la diferencia haciendo que éste último tenga lo mismo que el «HEAD». Por otro lado, la diferencia entre nuestro área de trabajo y el `stage` es que tenemos un archivo que no está en la siguiente versión, o lo que es lo mismo, es un archivo “sin seguimiento”.

Como no queremos borrarlo, vamos a dehacer este cambio. ¿Sin pensar mucho, qué propone? ... lo más intuitivo es hacer el `reset` porque es una forma de eliminar ese borrado pendiente de la zona de `stage` pero escribimos menos si hacemos:

```
prompt: ~/mensajes> git add saludar.cpp
```

que iguala las tres zonas de trabajo y por tanto deja el proyecto como estaba.

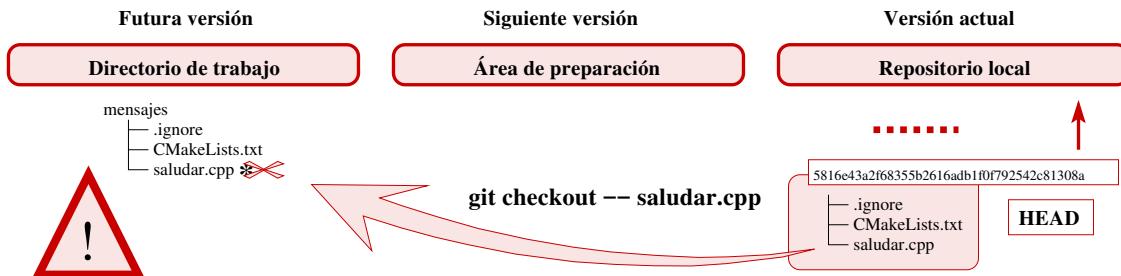
Recuperando la versión anterior de un archivo

En la sección anterior hemos deshecho la operación de añadido en el área de preparación. También es posible deshacer los cambios que hemos realizado en nuestro directorio de trabajo. Por ejemplo, imagine que hemos modificado el archivo `saludar.cpp` cambiando varias cosas pero queremos rectificar y volver a la última versión que tenemos confirmada. Lo podemos hacer con⁵:

⁵Observe que el tercer argumento es un guion doble «`--`» independiente de los otros tres. La órdenes «Git» se componen de opciones y argumentos que pueden generar múltiples combinaciones. En general, puede interpretar este guion doble como una indicación de que no hay más opciones, que el resto son argumentos para la orden que se está ejecutando.

```
prompt: ~/mensajes> git checkout -- saludar.cpp
```

El resultado es que volvemos a tener la versión que hay en el «HEAD», sin modificar. Podríamos representarlo gráficamente como sigue:



donde volvemos a tener un archivo `saludar.cpp` sin modificaciones y por tanto un estado en el que no hay nada que añadir al `stage` ni confirmar.

Aunque recordemos que hemos presentado las operaciones para deshacer diferencias entre los dos primeros bloques o los dos segundos. Aquí se repite de nuevo la misma idea. Realmente la orden de recuperación de la versión anterior recupera el archivo de la versión anterior, el que hay en la zona de `stage`. Sin embargo, en el ejemplo que hemos puesto la versión anterior a la que estábamos editando en el directorio de trabajo es realmente la de «HEAD».

Si quiere, podemos representar la operación de una forma un poco más compleja, haciendo énfasis en la recuperación de las diferencias con la zona de `stage` como sigue:



donde incluso hemos presentado una situación donde habíamos preparado una primera modificación de `saludar.cpp` y habíamos editado una nueva versión del archivo. La rectificación con `checkout` nos permite recuperar la versión anterior, la que estaba preparada.

Una vez realizada esta rectificación, no hay diferencia con la zona de `stage`. El nuevo estado será que no hay nada para `commit` pero sí para `add`. Si vuelve a ejecutar la misma orden `checkout` obtendrá una nueva versión del archivo `saludar.cpp`, el que estaba confirmado en el «HEAD».

Finalmente, tenga en cuenta que estas operaciones le harán perder las modificaciones del archivo. Deberá realizarlas con cuidado. Más adelante veremos alguna versión más avanzada que le permite recuperar estados anteriores y que ya podemos adelantar que tendrán este tipo de advertencia, con un mayor énfasis si es posible.

C.3.2 Mostrando diferencias

Si observa los anteriores ejemplos —los gráficos que muestran la posibilidad de tener varias versiones, en el directorio de trabajo, en el `stage` o en la versión confirmada «HEAD»— es probable que lo primero que eche de menos es una forma de comparar las distintas versiones. La solución es la orden `diff` que permite comparar distintas versiones, entre directorio de trabajo, zona de preparación, anteriores versiones, etc.

La orden, como tantas otras, tiene múltiples opciones. En esta sección vamos a mostrar los usos más simples y habituales. Como anteriormente, lo hacemos con ejemplos concretos de nuestro proyecto. Suponga que estamos en un estado con los tres archivos en el directorio de trabajo y sin nada que confirmar:

```
prompt: ~/mensajes> git status
En la rama master
nothing to commit, working directory clean
```

Comenzamos modificando el archivo `CMakeLists.txt` cambiando algún identificador y añadiendo una línea para compilar con el estándar C++14. El nuevo archivo es el siguiente:

```

1 project ( MENSAJES )
2 cmake_minimum_required (VERSION 3.5)
3
4 set ( CMAKE_CXX_STANDARD 14)
5 set ( MENSAJES_SRCS
6     saludar.cpp
7     )
8 add_executable ( mensaje ${MENSAJES_SRCS} )

```

El estado del proyecto cambia, pues el archivo del directorio de trabajo está modificado con respecto a lo que hay en el repositorio. Podemos preguntarnos qué diferencias hay.

El formato de `diff`

Para consultar las diferencias entre el fichero modificado del directorio de trabajo y la versión anterior usamos la orden `diff`. Esta orden, sin argumentos adicionales, localiza los ficheros que difieren y muestra todas las diferencias. El resultado es el siguiente:

```

prompt:~/mensajes> git diff
diff --git a/CMakeLists.txt b/CMakeLists.txt
index 1e59e34..a795a9d 100644
--- a/CMakeLists.txt
+++ b/CMakeLists.txt
@@ -1,8 +1,8 @@
 project ( MENSAJES )
 cmake_minimum_required (VERSION 3.5)

-set ( MENSAJES_SOURCES
+set ( CMAKE_CXX_STANDARD 14 )
+set ( MENSAJES_SRCS
    saludar.cpp
)
-
-add_executable ( mensaje ${MENSAJES_SOURCES} )
+add_executable ( mensaje ${MENSAJES_SRCS} )

```

En nuestro caso, sólo hemos modificado el archivo `CMakeLists.txt`, el único que muestra. Hubiéramos obtenido el mismo resultado si indicamos explícitamente el nombre de este archivo: «`git diff CMakeLists.txt`».

El formato se compone de una cabecera que describe los archivos que se comparan y de una serie de trozos —del inglés *chunk*— que marcan las diferencias. Los archivos comparados se encabezan con `diff...` y cada trozo con `@@`. Cada línea contiene:

```

prompt:~/mensajes> git diff
diff --git a/CMakeLists.txt b/CMakeLists.txt # Ficheros a/b comparados
index 1e59e34..a795a9d 100644           # hash y tipo archivo
--- a/CMakeLists.txt                      # a: con signos ---
+++ b/CMakeLists.txt                      # b: con signos ===
@@ -1,8 +1,8 @@
# Se muestra a: desde línea 1, 8 líneas (b coincide)
 project ( MENSAJES )                     # Sin signos... coinciden
 cmake_minimum_required (VERSION 3.5)       # Sin signos... coinciden
# Sin signos... coinciden
-set ( MENSAJES_SOURCES                   # Esto está en el a (---)
+set ( CMAKE_CXX_STANDARD 14 )              # Esto está en el b (++)
+set ( MENSAJES_SRCS                      # Esto está en el b (++)
    saludar.cpp                           # Sin signos... coinciden
)
# Sin signos... coinciden
-
# Esto está en el a (---)
-add_executable ( mensaje ...            # Esto está en el a (---)
+add_executable ( mensaje ${MENS...        # Esto está en el b (++)

```

Si piensa en el estado en que está el proyecto, se dará cuenta que esta es la diferencia entre el directorio de trabajo y la versión anterior, la que corresponde a «`HEAD`». Recuerde la sección anterior, cuando no tenemos el archivo en el `stage`, la versión anterior es la que está en el «`HEAD`».

Directorio de trabajo vs “siguiente versión” vs HEAD

La situación normalmente no es tan simple con un único archivo, sino que normalmente podemos tener varios archivos modificados e incluso algunos en la zona de preparación. Para mostrar qué ocurre cuando tenemos tres versiones del archivo, vamos a pasar el archivo modificado a la zona de preparación con «`add`» y además modificamos el que está en el directorio de trabajo. Por ejemplo, hacemos que la versión de CMake mínima sea la versión 3.0. Finalmente, cambiamos de nuevo el archivo `saludar.cpp` haciendo que el mensaje que escribe tenga un único punto. De esta forma, tenemos el siguiente estado:

```
Consola

prompt: ~/mensajes> git status
En la rama master
Cambios para hacer commit:
  (use «git reset HEAD <archivo>...» para sacar del stage)

    modificado:      CMakeLists.txt

Cambios no preparados para el commit:
  (use «git add <archivo>...» para actualizar lo que se confirmará)
  (use «git checkout -- <archivo>...» para descartar cambios en el directorio de trabajo)

    modificado:      CMakeLists.txt
    modificado:      saludar.cpp
```

En la que tenemos tres versiones de un archivo y dos del otro. Gráficamente lo podemos representar así:



La situación parece más complicada, pero la orden `diff` anterior vuelve a hacer lo mismo, comparar nuestra versión del directorio de trabajo con la anterior. El resultado ahora puede resultar extraño, porque aparecen los dos archivos en la lista:

```
Consola

prompt: ~/mensajes> git diff
diff --git a/CMakeLists.txt b/CMakeLists.txt
index a795a9d..b5fb40c 100644
--- a/CMakeLists.txt
+++ b/CMakeLists.txt
@@ -1,5 +1,5 @@
 project ( MENSAJES )
-cmake_minimum_required (VERSION 3.5)
+cmake_minimum_required (VERSION 3.0)

set ( CMAKE_CXX_STANDARD 14 )
set ( MENSAJES_SRCS
diff --git a/saludar.cpp b/saludar.cpp
index 4cccbc9..52c32df 100644
--- a/saludar.cpp
+++ b/saludar.cpp
@@ -3,5 +3,5 @@ using namespace std;

int main()
{
- cout << "Comenzamos la sesión..." << endl;
+ cout << "Comenzamos la sesión." << endl;
}
```

de forma que el primero se ha comparado con la versión del `stage` y el segundo con la de «`HEAD`». De nuevo, volvemos a recordar que estamos comparando con la versión anterior, es decir, la del `stage`. Si un archivo no está en esa zona es porque es el mismo que hay en el «`HEAD`».

Pero tal vez está interesado en saber la diferencia entre el **stage** y la versión confirmada del «**HEAD**». Se puede consultar añadiendo una opción a la orden⁶:

Consola

```
prompt:~/mensajes> git diff --cached
... lo que se presentó más arriba...
```

que obtiene la salida que vimos más arriba, cuando hicimos la primera consulta de diferencias entre esas dos versiones. Es interesante destacar que en esta consulta no aparece nada sobre **saludar.cpp**, puesto que no hay diferencias entre la versión en el **stage** y el «**HEAD**».

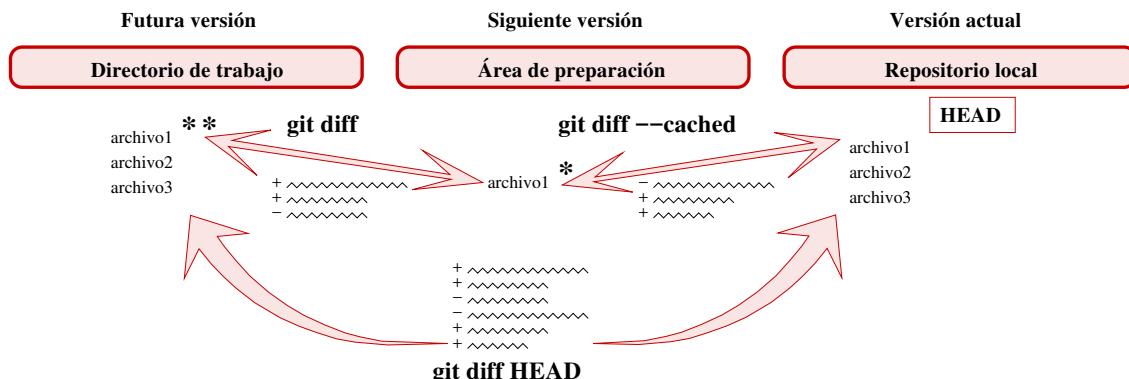
Pero tal vez le interesa la diferencia entre el directorio de trabajo y el «**HEAD**» que para el caso será la versión final que va a confirmar. Si queremos más detalles sobre esta diferencia para el archivo **CMakeLists.txt** podemos usar:

Consola

```
prompt:~/mensajes> git diff HEAD CMakeLists.txt
... la diferencia incluyendo todos los cambios
```

Recuerde que si no hubiéramos especificado el archivo, hubiera presentado la diferencia también del archivo **saludar.cpp** que es distinto al que está en el «**HEAD**».

Si revisa las distintas alternativas, hemos presentado tres posibilidades para poder comparar distintas áreas de almacenamiento. Gráficamente, lo podemos resumir como sigue:

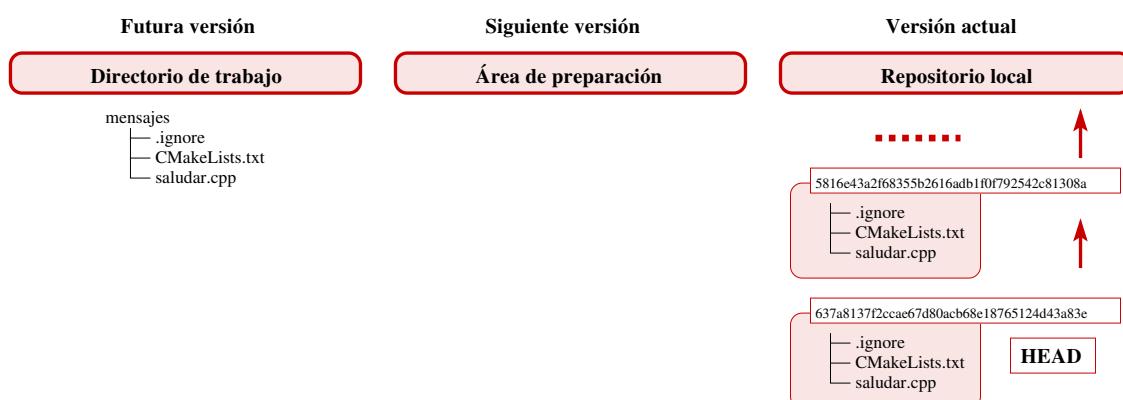


Una vez inspeccionados todos los cambios, podemos estar convencidos para generar una nueva versión de nuestro proyecto. Lo hacemos directamente en una sola línea:

Consola

```
prompt:~/mensajes> git commit -a -m "Cmake 3.0 y C++ 14"
[master 637a813] Cmake 3.0 y C++ 14
 2 files changed, 5 insertions(+), 5 deletions(-)
```

lo que finalmente nos lleva a un nuevo estado con una nueva versión confirmada:



⁶Tambien se puede usar **staged** en lugar de **cached**.

C.3.3 Historial del repositorio

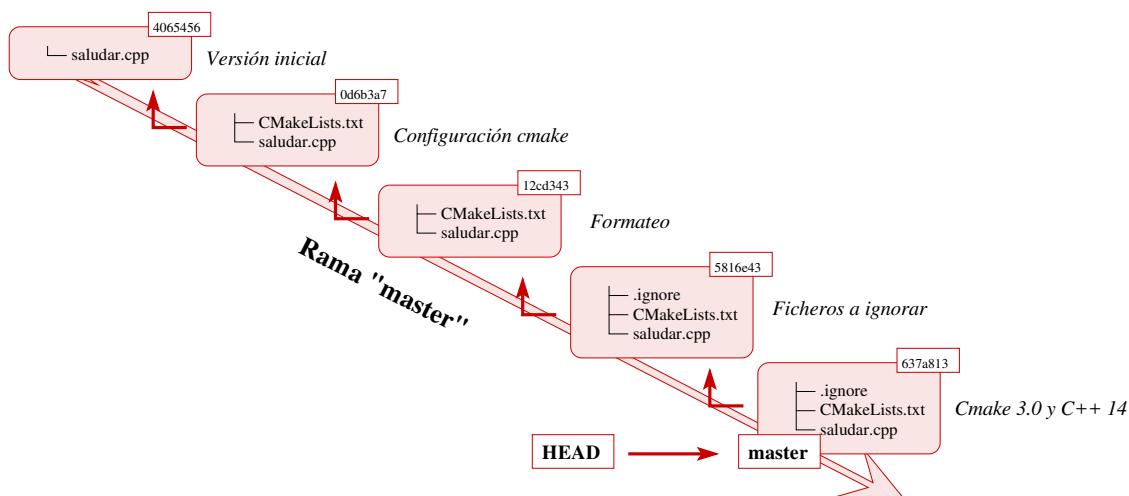
En las secciones anteriores hemos estado interactuando entre el directorio de trabajo, el **stage** y la última confirmación «**HEAD**». El repositorio contiene todo el historial de nuestro proyecto, documentando y con todos los detalles, incluyendo autor del cambio, fechas, archivos, etc.

Es importante entender en qué consiste el historial del repositorio, no sólo por recuperar esos datos, sino porque son el primer paso para entender las ramificaciones y el modelo de interacción de grupos de programadores en un mismo proyecto; probablemente, una de las grandes fortalezas de «**Git**».

En este tema, vamos a considerar una única rama de desarrollo llamada «**master**» aunque nosotros no la manejamos con ese nombre, sino con «**HEAD**» que hemos usado como el punto donde estamos situados en el historial. Para representar la situación actual en este punto del proyecto, vamos a modificar ligeramente los detalles:

- Usaremos nombres cortos para el **hash**. Es habitual que el usuario use los primeros dígitos en lugar de todo el número. Por defecto, «**Git**» usará los 7 primeros números, aunque es importante destacar que éstos deben ser únicos para que no haya ambigüedad cuando se use en lugar de la versión original de 40 caracteres. Por tanto, «**Git**» podría usar más de 7 si es necesario.
- El nombre «**HEAD**» apunta a **master**. En principio usaremos el nombre «**HEAD**» que siempre nos va a apuntar al último **commit** que hemos hecho. En realidad no es más que un puntero que indica dónde estamos situados; es conveniente ir acostumbrándonos a ello.

Con estos detalles, nuestra situación en el historial se puede presentar como sigue:



En esta sección vamos a incluir algunos detalles de cómo consultar el historial y describir algunas operaciones relacionadas. Más adelante, cuando estudie la gestión de ramas ampliará estos conocimientos con nuevas operaciones y muchas más posibilidades.

Consultando el historial

La forma más simple para consultar el historial es la orden **log** que obtiene en la salida información sobre los **commit's** que se han realizado, el autor, la fecha y el comentario. Un ejemplo con nuestro proyecto es:

```

Consola

prompt:~/mensajes> git log
commit 637a8137f2ccae67d80acb68e18765124d43a83e
Author: Antonio Garrido <agarrido@decsai.ugr.es>
Date:   Fri Dec 30 13:53:03 2016 +0100

        Cmake 3.0 y C++ 14

commit 5816e43a2f68355b2616adb1f0f792542c81308a
Author: Antonio Garrido <agarrido@decsai.ugr.es>
Date:   Fri Dec 23 12:53:12 2016 +0100

        Ficheros a ignorar

.... otras entradas hasta el primero...

```

donde puede ver que los más recientes son los primeros; probablemente los que más le interesen. Aunque si quisiera consultar más entradas, la salida por defecto podría resultar muy larga.

Las opciones para obtener distintos listados son muy amplias, incluyendo la posibilidad de configurar los campos que se muestran o limitando la salida con condiciones sobre el autor o que contiene cierta cadena. Por ejemplo, una forma habitual que uso para consultar el historial es la siguiente:

```
prompt:~/mensajes> git log --oneline --decorate --graph
* 637a813 (HEAD -> master) Cmake 3.0 y C++ 14
* 5816e43 Ficheros a ignorar
* 12cd343 Formateo
* 0d6b3a7 Configuración cmake
* 4065456 Versión inicial
```

donde las opciones me permiten, respectivamente, escribir cada entrada en una línea, incluir detalles de las etiquetas **HEAD/master** y —aunque ahora no se aprecia la ventaja— añadir caracteres ASCII que representen un gráfico de las ramas que contiene el historial.

En este punto, es un momento ideal para presentar los **alias** que «**Git**» admite para simplificar las órdenes. Seguramente conoce los alias del sistema; es algo similar. Básicamente, la idea es crear un nuevo nombre que equivale a cierta orden y parámetros de «**Git**». Por ejemplo, en mi caso he creado un alias con esta línea:

```
prompt:~/mensajes> git config --global alias.log '--oneline --decorate --graph'
```

lo que me permite usar «**git l**» para obtener el mismo resultado anterior. Incluso puedo añadir alguna opción adicional:

```
prompt:~/mensajes> git l -3
* 637a813 (HEAD -> master) Cmake 3.0 y C++ 14
* 5816e43 Ficheros a ignorar
* 12cd343 Formateo
```

para consultar las últimas 3 entradas del **log**. O si quiere puede consultar las entradas que han afectado a cierto **path**, por ejemplo:

```
prompt:~/mensajes> git l saludar.cpp
* 637a813 (HEAD -> master) Cmake 3.0 y C++ 14
* 12cd343 Formateo
* 4065456 Versión inicial
```

o incluso limitando la salida a un cierto rango de revisiones:

```
prompt:~/mensajes> git l 0d6b3a7..637a813
* 637a813 (HEAD -> master) Cmake 3.0 y C++ 14
* 5816e43 Ficheros a ignorar
* 12cd343 Formateo
```

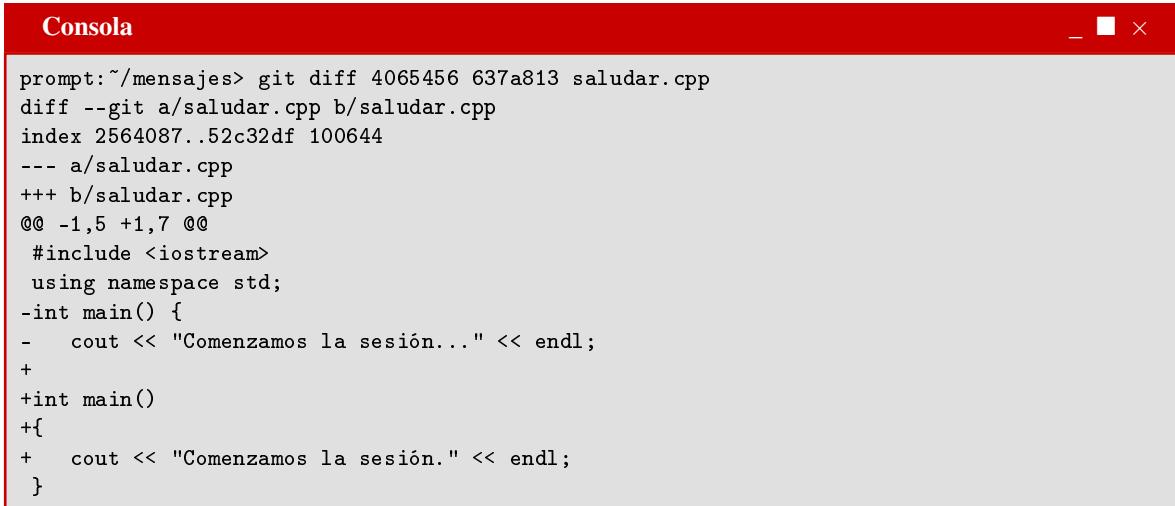
donde puede ver que el formato de un rango es una referencia a las dos revisiones que delimitan el rango del historial, separadas por dos puntos. Desde «**0d6b3a7**» (que no se incluye) hasta «**637a813**», incluida.

Las posibilidades se multiplican, es un buen momento para tener en cuenta algunas herramientas gráficas que le permiten ver el historial de una forma mucho más cómoda. Existen múltiples programas⁷ aunque tal vez una habitual en los tutoriales es «**gitk**», un programa que representa el historial del repositorio. Normalmente se instala con «**Git**» así que puede usarlo para ojear qué tipo de presentación puede obtener.

⁷Por ejemplo *git-cola*, *gitg*, *gitk*, *qgit*, etc.

Diferencias entre dos versiones

Las diferencias no se limitan a las 3 zonas de almacenamiento. También se pueden realizar entre revisiones que se han almacenado en el historial. Sólo será necesario incluir qué versiones queremos comparar y, si lo desea, el archivo que queremos consultar. Por ejemplo, en nuestro caso podemos hacer:

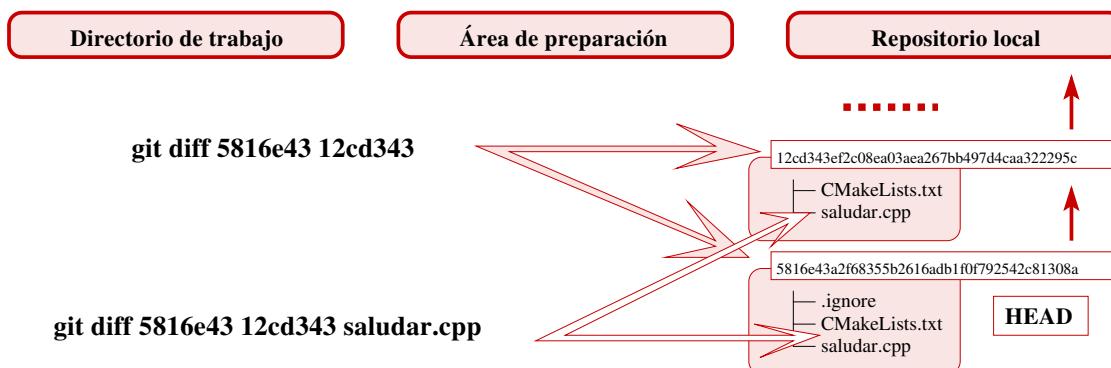


```

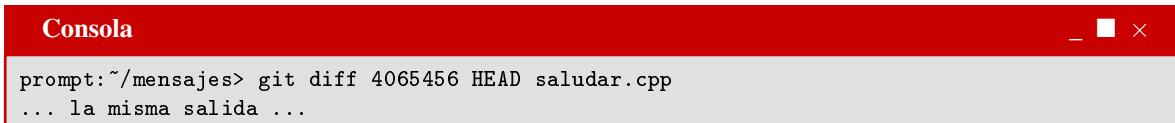
prompt:~/mensajes> git diff 4065456 637a813 saludar.cpp
diff --git a/saludar.cpp b/saludar.cpp
index 2564087..52c32df 100644
--- a/saludar.cpp
+++ b/saludar.cpp
@@ -1,5 +1,7 @@
 #include <iostream>
 using namespace std;
-int main() {
-    cout << "Comenzamos la sesión..." << endl;
+
+int main()
+{
+    cout << "Comenzamos la sesión." << endl;
}

```

que consulta la diferencia entre la primera y la última versión del archivo `saludar.cpp`⁸. Esta comparación entre versiones del repositorio se puede expresar gráficamente así:



Aunque los números son un poco incómodos. Sabemos que la última versión es «`HEAD`», así que podemos conseguir lo mismo con:



```

prompt:~/mensajes> git diff 4065456 HEAD saludar.cpp
... la misma salida ...

```

Es mucho más fácil usar etiquetas en lugar de números. «Git» facilita estas referencias: podemos usar «`HEAD`» —o incluso `master`, que es lo mismo— pero también referenciar los padres con el carácter `^`. Por ejemplo, si queremos la diferencia entre la versión del «`HEAD`» y la anterior, podemos añadir este carácter para indicar el padre⁹:



```

prompt:~/mensajes> git diff HEAD HEAD^ saludar.cpp
...

```

e incluso padre de padre añadiendo varios caracteres `^`; aunque en caso de especificar varios pasos es más sencillo la sintaxis:



```

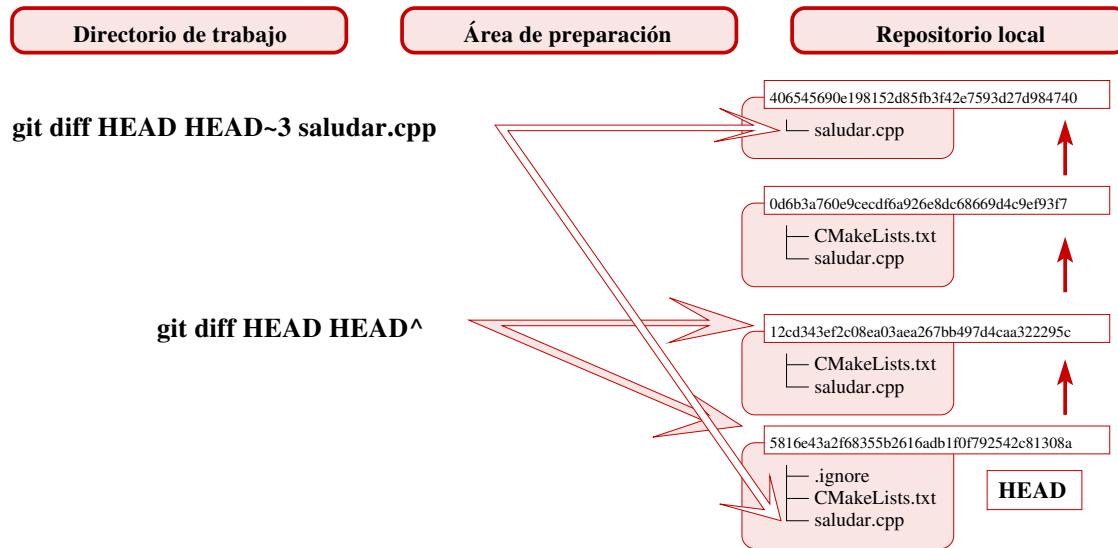
prompt:~/mensajes> git diff HEAD HEAD~4 saludar.cpp
...

```

⁸Sí, mucho «Git» y poco C++, esa era la idea.

⁹ Cuando estudie ramas verá que también podemos especificar con un número adicional qué parent es al que nos referimos, pues si unimos dos ramas, una revisión podrá tener 2 padres.

que muestra la diferencia con la versión 4 ancestros hacia atrás, la que corresponde a la versión inicial. Una versión gráfica de esta representación es:



Rectificando el último commit

«Git» incluye operaciones que permiten modificar el historial en gran medida. Se puede considerar un tema avanzado que no tiene sentido en este lugar. Sin embargo, es interesante incluir una operación especial que tal vez pueda surgir incluso en un nivel básico: rectificar el último **commit**. Nos referimos a que tal vez haya terminado de realizar una confirmación y se da cuenta de que debería cambiar algún detalle, añadiendo cierto archivo, cambiando el mensaje asociado, etc.

Si acaba de realizar una confirmación y quiere cambiar el mensaje, sólo tiene que hacer:

```
Consola
prompt: ~/mensajes> git commit --amend
```

que como no se ha cambiado nada en el directorio de trabajo ni en el **stage**, simplemente lanzará el editor del mensaje para cambiar la documentación asociada a la confirmación.

Pero además podría cambiar los archivos relacionados. Si ha realizado un **commit** y cambia algunas cosas, puede añadirlas al **stage** para que, al realizar esta rectificación, se actualice la revisión. Gráficamente lo podemos representar así:



Más adelante estudiará otras posibilidades mucho más potentes, aunque también de alguna forma peligrosas, pues está rectificando el historial del repositorio. En cualquier caso, esta operación de rectificación para resolver un error que acabamos de cometer es algo simple y eficaz para resolver esa probable situación.

Volviendo a una versión anterior

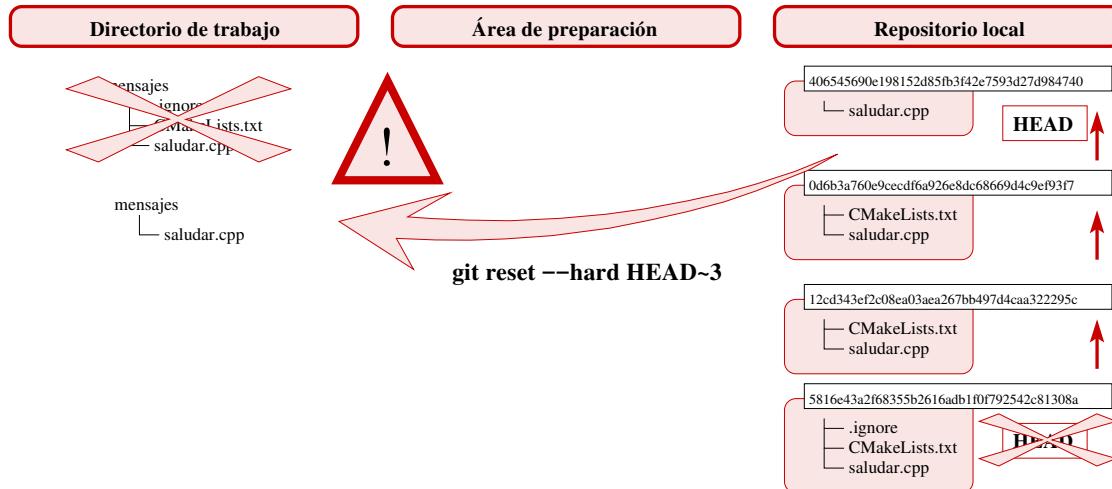
Existe la vuelta atrás en el tiempo. Es una operación que presentamos para —aunque resulte extraño— recomendar que no se use. Sin embargo, es algo especialmente ilustrativo para conocer cómo «Git» nos permite no sólo saber lo que hicimos, sino incluso recuperar exactamente el mismo estado.

La forma de hacerlo es mediante una orden conocida aunque con una opción que la hace peligrosa. La orden es:

Consola

```
prompt: ~/mensajes> git reset --hard 4065456
HEAD is now at 4065456 Versión inicial
```

que provoca un resultado como el que representamos a continuación:



Hemos movido **Head** pero además hemos sobreescrito nuestro directorio de trabajo. Si teníamos algún archivo con alguna modificación que no habíamos salvado al repositorio, ya no lo podemos recuperar. Para deshacerlo, tal vez piense en repetir la orden de **reset** indicando que se mueva a **Head**, pero no haría nada, ya estamos en **Head**, de hecho, siempre estamos en **Head**. Si hace un nuevo **commit**, su padre será **4065456**. Observe el resultado de preguntar por el historial:

Consola

```
prompt: ~/mensajes> git log --oneline --decorate
4065456 (HEAD -> master) Versión inicial
```

Estamos al principio, no hay más que una confirmación. Si realmente queremos “*volver al presente*”, podemos hacer:

Consola

```
prompt: ~/mensajes> git reset --hard 12cd343
HEAD is now at 12cd343 Formateo
```

que de nuevo recupera nuestro estado original, aunque cuidado: el estado original de la última versión confirmada.

Esto nos permite reflexionar sobre la orden **reset**, que se presentó de forma inofensiva. Por defecto, esta orden repone el contenido del **stage** sin tocar el directorio de trabajo. Además, si especificamos cierto archivo no mueve «**HEAD**». Todo lo será más sencillo cuando estudio las ramas y cómo «**HEAD**» puede saltar a distintos puntos del historial.

Ahora mismo, use **reset** como habíamos presentado en las secciones anteriores, aunque sirva este ejemplo para mostrar el potencial de **Git** así como algún detalle adicional sobre **reset**.

C.4 Repositorios remotos

El control de versiones con «**Git**» permite un modelo de trabajo distribuido, con múltiples desarrolladores trabajando independientemente, sobre sus repositorios locales, pero con la capacidad de compartir y mezclar el trabajo en un proyecto común. Con los repositorios remotos, abrimos la puerta a esta forma de colaboración.

En principio, podríamos tener un grupo de programadores trabajando conjuntamente con nosotros de forma que podríamos intercambiar información con cada uno de ellos de forma independiente. Si bien esto es posible e incluso a cierto nivel práctico¹⁰, es una buena idea mantener un repositorio principal, un lugar donde se encuentra la última versión confirmada y que sirve para coordinar a todo el grupo de programadores. De esta forma, se puede desarrollar alguna nueva capacidad del software de forma local o compartiendo avances entre un grupo de programadores para finalmente confirmar el trabajo sobre el repositorio principal, compartiéndolo con todos.

En general, podemos disponer en nuestro directorio local de múltiples ramas, con varios directorios remotos —unos de sólo lectura y otros también de escritura— también con varias ramas. Sin embargo, esta situación algo más complicada la dejamos para más adelante.

¹⁰Por ejemplo, dos personas están trabajando en una futura revisión y comparten sus repositorios para que el otro pueda acceder a los últimos cambios.

En esta sección vamos a presentar las órdenes básicas para interactuar sobre un repositorio principal que contiene también una rama *master* como la nuestra, desde el que descargamos lo que otros desarrollan y sobre el que actualizamos el software con nuestras aportaciones. Tenga en cuenta que aunque no lo contamos todos, abarca las principales operaciones, lo que permite empezar a trabajar y practicar con «[Git](#)».

C.4.1 Protocolos del servidor

Un repositorio en un servidor no es más que un lugar donde se almacenan los datos de forma similar a lo que se hace en su repositorio local. El hecho de que esté en un servidor implica que el acceso a esos datos se tendrá que realizar con cierto protocolo, dependiendo de la configuración del servidor. Protocolos disponibles son:

- *Local*. Podemos usar un directorio local. Por ejemplo, podría configurar un directorio compartido que monta en su máquina y que contiene el repositorio accesible para varios desarrolladores.
- *SSH*. Se basa en la autenticación *SSH*, que resulta muy cómodo porque es fácil de configurar y fácil de gestionar para la autenticación de los usuarios. Si no quiere accesos anónimos al servidor, es una opción muy eficaz.
- *HTTP*. Se implementa aprovechando el protocolo *HTTP/S*. Es muy frecuente; si revisa algunos proyectos en la red, encontrará que los enlaces para descargar los fuentes se basan en este protocolo; probablemente, por su facilidad para configurar accesos anónimos.
- *GIT*. Es un protocolo especial para «[Git](#)». Es menos habitual porque no usa autenticación.

No es necesario explicar cómo funcionan estos protocolos para poder manejar correctamente «[Git](#)». Si está interesado, puede consultar más detalles en Chacon[14].

Para este tutorial, lo importante es entender que nuestro servidor es un lugar de almacenamiento donde se guarda el repositorio; que puede darnos accesos de lectura o lectura/escritura; y que «[Git](#)» se encargará de resolver el problema de la conexión y el protocolo. En concreto, disponemos de la *url* de nuestro servidor, donde tendremos permisos de lectura y escritura para poder traer nuevas versiones así como incorporar nuestros cambios.

Ejemplo: un remoto por ssh

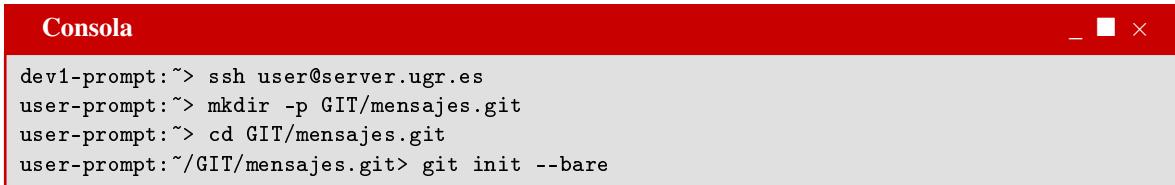
En esta sección vamos a crear un repositorio remoto al que accederemos con el protocolo *ssh*. En este manual, no estamos interesados en que aprenda a gestionar servidores con repositorios, pero servirá para que entienda mejor en qué consiste. Ya conoce el ejemplo que hemos preparado en las secciones anteriores, ahora lo llevaremos a la red.

Para crear el servidor, suponga ya configurado el acceso *ssh* a un servidor en la cuenta user@server.ugr.es. Además podemos suponer varios desarrolladores en distintas máquinas para acceder a esta cuenta, todos con permiso de lectura y escritura. Tenemos dos posibilidades:

- Crear un repositorio vacío en el servidor. Básicamente es lo que hemos hecho antes con la orden [init](#), pero en el directorio del servidor que contendrá el repositorio.
- Crear un repositorio con una versión inicial del proyecto. Es lo que haremos con el proyecto que hemos presentado en las secciones anteriores.

El contenido del servidor y nuestro directorio local son muy parecidos. En nuestra máquina tenemos que mantener el repositorio, pero también un directorio de trabajo. De ahí que hayamos mantenido los archivos accesibles directamente mientras el repositorio se gestionaba de forma oculta en un directorio [.git](#) no visible. Básicamente, el servidor no tiene directorio de trabajo, por lo que basta con lo que tenemos en el directorio [.git](#), sin necesidad de ocultarlo.

En el primer caso, podríamos hacer algo así:



```
dev1-prompt:~> ssh user@server.ugr.es
user-prompt:~> mkdir -p GIT/mensajes.git
user-prompt:~> cd GIT/mensajes.git
user-prompt:~/GIT/mensajes.git> git init --bare
```

donde puede ver que tenemos acceso al servidor sin necesidad de autenticarse y que el repositorio será muy similar a lo que hicimos en nuestra máquina, aunque lo hemos nombrado con la extensión [.git](#) que es habitual. Lo más interesante es la opción [--bare](#)¹¹ que indica que sea un repositorio no oculto, puesto que no necesita directorio de trabajo.

Más ilustrativo es cómo se puede crear con contenido. Volvemos a nuestro directorio y ejecutamos:



```
prompt:~/mensajes> cd ..
prompt:~> git clone --bare mensajes/.git mensajes.git
Cloning into bare repository 'mensajes.git'...
hecho.

prompt:~> scp -r mensajes.git user@server.ugr.es:/home/user/GIT
```

¹¹Además se puede usar [-shared](#) si quiere que dé permiso de escritura de grupo para otros usuarios que puedan acceder al directorio.

donde la parte más relevante está en clonar el repositorio. Básicamente, copia el directorio `.git` incluyendo subdirectorios. Note que hemos supuesto el directorio **GIT** en el servidor y que hemos configurado el acceso sin necesidad de introducir ninguna clave. Por último, puede borrar el directorio `mensajes.git`.

Ahora sólo necesita comunicar a otros desarrolladores con permiso para acceder al repositorio la `url` desde donde descargar su primera versión del proyecto.

C.4.2 El repositorio `origin`

En la primera parte de este tutorial hemos creado un repositorio local —con la orden `init`— de forma que trabajamos sin directorio remoto. Otra forma de empezar a trabajar en un proyecto es descargando un repositorio remoto. Es posible que tal vez lo haya hecho, pues los repositorios «**Git**» de proyectos de código abierto que existen en la red dan instrucciones muy concretas para crear un duplicado de sus contenidos: la orden `clone`, probablemente con una `url` basada en el protocolo `http/s`.

Suponga que dos colaboradores —`dev1` y `dev2`— van a trabajar con nuestro mismo proyecto `mensajes`. Le hemos dado la `url` y configurado el acceso a la misma cuenta `user`. El primero de ellos descarga la versión actual en su directorio de proyectos C++ con la orden `clone`:

```
Consola
dev1-prompt:~/cpp> git clone user@server.ugr.es:GIT/mensajes.git
Clonar en «mensajes»...
remote: Counting objects: 17, done.
remote: Compressing objects: 100% (16/16), done.
remote: Total 17 (delta 1), reused 0 (delta 0)
Receiving objects: 100% (17/17), done.
Resolving deltas: 100% (1/1), done.
Comprobando la conectividad... hecho.
```

con lo que ya tiene una copia de todo el contenido en su directorio de trabajo `mensajes`. Pero realmente no tiene exactamente lo mismo que teníamos en nuestra máquina, puesto que esta clonación se ha realizado desde cierto repositorio remoto. En concreto, dispone de:

- Una copia del repositorio; esto ya lo habrá adivinado. Su directorio de trabajo contiene los mismos archivos y puede consultar la misma lista de revisiones.
- La configuración de un repositorio remoto, concretamente con nombre «`origin`» que “asocia” su repositorio local con el lugar de donde lo descargó.

Si el usuario `dev1` pregunta por la lista de repositorios remotos obtendrá:

```
Consola
dev1-prompt:~/cpp> cd mensajes
dev1-prompt:~/cpp/mensajes> git remote
origin
```

lo que indica que dispone del repositorio con nombre `origin`. El nombre no es especialmente relevante, se podría llamar de otra manera, incluso podría crear otro nombre en su lugar si no le gusta, pero es habitual al ser el nombre por defecto del directorio desde donde se clonó. Más interesante si añadimos una opción para más detalles:

```
Consola
dev1-prompt:~/cpp/mensajes> git remote -v
origin user@server.ugr.es:GIT/mensajes.git (fetch)
origin user@server.ugr.es:GIT/mensajes.git (push)
```

donde ya informa de que `origin` tiene asociada una `url` para “descargar” y otra para “subir” contenido.

Seguimiento de la rama `master`

No sólo tenemos conocimiento del lugar desde donde hemos clonado el repositorio. Además, «**Git**» se ha encargado de “asociar” las ramas `master`. Es decir, ha establecido que nuestra rama local `master` y la del servidor están relacionadas de forma que nuestra intención es descargar contenidos de `origin/master` a nuestra local y subir nuestras aportaciones desde la local a `origin/master`.

Esta información la puede obtener si pedimos detalles sobre el remoto `origin`:

```
dev1-prompt:~/cpp/mensajes> git remote show origin
* remote origin
  Fetch URL: user@server.ugr.es:GIT/mensajes.git
  Push URL: user@server.ugr.es:GIT/mensajes.git
  HEAD branch: master
  Remote branch:
    master tracked
  Local branch configured for 'git pull':
    master merges with remote master
  Local ref configured for 'git push':
    master pushes to master (up to date)
```

donde puede ver que la rama *master* está siendo seguida —del inglés *tracked*— y que cuando usamos las órdenes **pull/push** (lo veremos en breve) nuestra *master* corresponde a la remota.

Aunque le parezca muchos detalles, no se preocupe demasiado. Recuerde que se ha realizado de forma automática cuando hemos clonado. Además, no vamos a cambiar esta configuración.

Configuramos el *origin* en nuestra máquina

Esta sección le servirá para completar los comentarios, aunque tiene como objetivo ilustrar las diferencias con el repositorio descargado con **clone** para que entienda mejor lo que ha ocurrido.

En nuestra máquina no hay remotos. Nos hemos encargado de clonar nuestro directorio a un servidor, pero no ha cambiado nada. Lo podemos comprobar con:

```
prompt:~/mensajes> git remote -v
prompt:~/mensajes>
```

No es lo que queremos, porque también estamos interesados en interactuar con el repositorio remoto. Lo más sencillo es dejar esa primera versión y llamar a la clonación del remoto. De esa forma obtendremos lo mismo que **dev1**, con el remoto y el seguimiento configurado.

Si está empezando con «**Git**», puede saltar el resto de este apartado. Sin embargo, es posible que tenga curiosidad por saber si podemos configurar nuestro repositorio local. Efectivamente, podemos añadir ese seguimiento. En concreto, hacemos:

```
prompt:~/mensajes> git remote add origin user@server.ugr.es:GIT/mensajes.git
prompt:~/mensajes> git fetch
  From user@server.ugr.es:GIT/mensajes
    * [new branch]      master      -> origin/master
prompt:~/mensajes> git branch -u origin/master
Branch master set up to track remote branch master from origin.
```

que básicamente hace: añadimos un remoto con nombre **origin**, traemos los contenidos del remoto a nuestra máquina y le decimos que nuestro *master* sigue a **origin/master**.

C.4.3 Avanzando con el repositorio local

Una vez los desarrolladores tienen el repositorio actualizado, pueden trabajar en sus aportaciones. Sin entrar en posibles modelos de trabajo, vamos a suponer que hay tres desarrolladores: **Antonio**, **dev1** y **dev2**. Cada uno tenemos la oportunidad de usar las órdenes que hemos visto anteriormente para añadir nuevas versiones a nuestro repositorio local.

Programador 1: Cambio en el espacio de nombres

El programador **dev1** piensa que lo innovador del programa terminará por “internacionalizarlo”. Piensa que mejor no abrir el espacio **std** para evitar choques con identificadores en inglés. Ha decidido que lo mejor es evitar **using**. Elimina la línea y modifica el programa como sigue:

```
#include <iostream>

int main()
{
    std::cout << "Comenzamos la sesión." << std::endl;
```

Después del cambio, lo añade a su repositorio local como una nueva versión:

```
dev1-prompt:~/cpp/mensajes> git commit -a -m "Evitamos abrir std"
[master 803187a] Evitamos abrir std
1 file changed, 1 insertion(+), 2 deletions(-)
```

En este punto es interesante consultar el estado del repositorio local. Observe el `log` con las etiquetas añadidas:

```
dev1-prompt:~/cpp/mensajes> git log --oneline --decorate
803187a (HEAD -> master) Evitamos abrir std
637a813 (origin/master, origin/HEAD) Cmake 3.0 y C++ 14
5816e43 Ficheros a ignorar
12cd343 Formateo
0d6b3a7 Configuración cmake
4065456 Versión inicial
```

El repositorio local ha añadido la versión `803187a`, como habíamos visto antes, en su rama `master` y ha avanzado «`HEAD`» a la nueva versión. Por otro lado, también sabe dónde está el remoto y sus etiquetas.

Programador 2: Tiquismiquis ortográfico

El programador `dev2` está interesado en cambiar el mensaje de comienzo, lo que hace modificando la línea del programa y generando la versión:

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

int main()
{
    cout << "Comenzamos la sesión..." << endl;
}
```

Después del cambio, lo añade a su repositorio local como una nueva versión:

```
dev2-prompt:~/proyectos/mensajes> git commit -a -m "... pausa que expresa suspense"
[master bdcc027] ... pausa que expresa suspense
1 file changed, 1 insertion(+), 1 deletion(-)
```

En este punto es interesante consultar el estado del repositorio local. Observe el `log` con las etiquetas añadidas:

```
dev2-prompt:~/proyectos/mensajes> git log --oneline --decorate
bdcc027 (HEAD -> master) ... pausa que expresa suspense
637a813 (origin/master, origin/HEAD) Cmake 3.0 y C++ 14
5816e43 Ficheros a ignorar
12cd343 Formateo
0d6b3a7 Configuración cmake
4065456 Versión inicial
```

Ahora los dos desarrolladores están avanzando de forma independiente. Cada uno está creando su propia historia del proyecto, con cambios locales que el otro no conoce. Los dos están haciendo que la rama `master` avance, junto con su etiqueta «`HEAD`» que usamos para saber dónde estamos.

C.4.4 Subiendo nuestros cambios

Cuando quieras hacer públicos tus cambios, haciéndolos definitivos en el historial común del proyecto, subes los cambios al repositorio `origin`. Realmente, no es la única forma, pues podrías subirlos a otros remotos —tras configurarlos— o incluso mandarlos como “*parches*” a colaboradores antes de incorporarlos. Nosotros nos centramos en la interacción por medio del repositorio común que hemos llamado `origin`.

El desarrollador 2 —entusiasmado por su aportación de puntos suspensivos— quiere subir la versión cuanto antes. Para eso necesita la orden `push`. Esta orden tiene dos parámetros opcionales, el *nombre del remoto* y el *nombre de la rama*. En nuestro caso, si no decimos nada se subirán los cambios hechos en `master` al repositorio `origin`. El resultado es:

```
dev2-prompt:~/proyectos/mensajes> git push
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 312 bytes | 0 bytes/s, done.
Total 3 (delta 2), reused 0 (delta 0)
To user@server.ugr.es:GIT/mensajes.git
 637a813..bdcc027  master -> master
```

aunque posiblemente la salida no será exactamente esa, pues también podrá añadir un mensaje de aviso sobre un cambio de comportamiento a partir de la versión 2.0 de «[Git](#)», ya que hemos hecho un [push](#) por defecto, sin indicar nada sobre remoto o rama. En principio, puede ignorar este mensaje.

El resultado lo podemos hacer explícito con la consulta del [log](#):

```
dev2-prompt:~/proyectos/mensajes> git log --oneline --decorate
bdcc027 (HEAD -> master, origin/master, origin/HEAD) ... pausa que expresa suspense
637a813 Cmake 3.0 y C++ 14
5816e43 Ficheros a ignorar
12cd343 Formateo
0d6b3a7 Configuración cmake
4065456 Versión inicial
```

Observe que el remoto ha avanzado también hasta la nueva versión local, que incluye nuestro cambio.

Subiendo a un remoto modificado

Por otro lado, el desarrollador 1 ha creado una versión unos segundos después y quiere que los demás conozcan sus cambios con respecto a su decisión sobre no abrir [std](#). Decide subir sus cambios con la orden [push](#), aunque a él le gusta especificar exactamente el remoto y la rama a subir:

```
dev1-prompt:~/cpp/mensajes> git push origin master
To user@server.ugr.es:GIT/mensajes.git
! [rejected]      master -> master (fetch first)
error: failed to push some refs to 'user@server.ugr.es:GIT/mensajes.git'
consejo: Updates were rejected because the remote contains work that you do
consejo: not have locally. This is usually caused by another repository pushing
consejo: to the same ref. You may want to first integrate the remote changes
consejo: (e.g., 'git pull ...') before pushing again.
consejo: See the 'Note about fast-forwards' in 'git push --help' for details.
```

El resultado ha sido que se ha rechazado la orden; no ha subido nada. El motivo es muy simple: el remoto tiene cambios confirmados que el desarrollador no tiene.

Si considera que el remoto debería aceptar las nuevas versiones, piense un instante en el problema de integrar los cambios de los dos desarrolladores. El remoto no sabe sumar cambios porque habrá casos en los que no sabrá qué hacer; como veremos, algunos cambios entran en conflicto. Lo que hay que hacer es descargar los cambios, integrarlos en nuestro repositorio local y finalmente subirlos. En ese caso, el remoto no tiene que hacer nada especial, el problema de mezclar los cambios lo resolvemos nosotros.

C.4.5 Descargando una nueva versión

El desarrollador 1 quiere subir sus cambios, pero antes tendrá que descargar lo que ya han hecho otros. La operación se realiza en dos pasos:

- *Traer lo que hay en el remoto.* La orden es [fetch](#) más el nombre del remoto (que es opcional). Lo que hace es traer todo lo que hay en ese remoto y que nosotros no tenemos.
- *Combinarlo con lo que tenemos.* La orden [fetch](#) no cambia nada en nuestro repositorio local, que sigue teniendo lo mismo. Sólo nos ha actualizado los contenidos que hay en el remoto. Si queremos combinarlo tenemos que usar la orden [merge](#).

Comenzamos actualizando lo que tiene el remoto:

```
dev1-prompt:~/cpp/mensajes> git fetch
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 2), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From user@server.ugr.es:GIT/mensajes
  637a813..bdcc027 master      -> origin/master
```

Observe que no ha sido necesario indicar que el remoto es `origin`. El resultado es que ya tenemos los contenidos. Podemos preguntarnos por si nuestro histórico local está en la misma situación. Hacemos `log`:

```
dev1-prompt:~/cpp/mensajes> git log --oneline --decorate
803187a (HEAD -> master) Evitamos abrir std
637a813 Cmake 3.0 y C++ 14
5816e43 Ficheros a ignorar
12cd343 Formateo
0d6b3a7 Configuración cmake
4065456 Versión inicial
```

Estamos exactamente en la misma situación. Incluso podríamos seguir creando nuevas versiones sin cambiar nada más. Si observa detenidamente, realmente no ha salido lo mismo, porque ahora no aparecen las etiquetas del remoto. Es exactamente lo que tiene que salir, pues el `log` nos presenta la posición del «`HEAD`» y sus ancestros. En ninguno de ellos están situadas las etiquetas del remoto.

Mezclando

Para poder integrar los resultados después de `fetch`, usamos la orden `merge`. Esta orden se encarga de mezclar las dos versiones. Funciona realmente bien, pues si hay cambios en archivos independientes, los mezcla sin problemas y cuando hay cambios en un mismo archivo, usa el archivo original común y los dos nuevos archivos e intenta realizar una mezcla automática.

En la mayoría de los casos, la mezcla se realizará sin problemas, pues es probable que los desarrolladores estén tocando puntos distintos del programa. En nuestro ejemplo no es así, pues hemos tocado exactamente las mismas cosas a propósito. El resultado es el siguiente:

```
dev1-prompt:~/cpp/mensajes> git merge
Automezclado saludar.cpp
CONFLICTO (contenido): conflicto de fusión en saludar.cpp
Automatic merge failed; fix conflicts and then commit the result.
```

que corresponde al conflicto que hemos provocado en la línea con el mensaje de salida. Era de esperar, cada desarrollador ha hecho una aportación distinta, tendremos que decidir qué queremos finalmente. Por su parte, «`Git`» intenta hacernos las cosas más fáciles, así que nos ha cambiado el archivo `saludar.cpp` que contiene el conflicto añadiendo las diferencias que hay entre nuestro «`HEAD`» local y lo que hay en el remoto. En concreto, podría obtener algo así:

```
#include <iostream>

int main()
{
<<<<< HEAD
    std::cout << "Comenzamos la sesión." << std::endl;
=====
    cout << "Comenzamos la sesión..." << endl;
>>>>> refs/remotes/origin/master
}
```

Note que el resto del fichero se ha mezclado correctamente. En concreto, la línea `using` no aparece, pues no ha entrado en conflicto; algo relevante, pues si optáramos por dejar la versión con puntos suspensivos tendríamos que añadirla. Como puede ver, los conflictos los tenemos que solucionar nosotros, no se puede realizar en el servidor de forma automática. De hecho, la solución no es ninguna de las dos, sino una mezcla: añadir los puntos suspensivos y mantener el espacio de nombres `std` sin abrir. Pero antes de arreglarlo, hablemos de otras alternativas que pueden ayudarnos en conflictos más complicados.

Configurando la mezcla

En este punto es interesante indicar un par de opciones que podrían añadirse para que le resulte más fácil la mezcla de versiones. Son dos opciones de configuración que puede optar por ignorar, pero que es interesante que conozca.

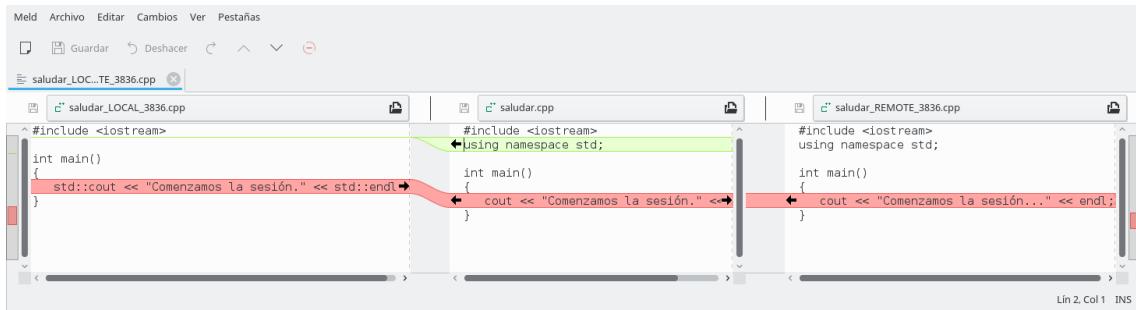
Le propongo que use las dos siguientes opciones para todos sus proyectos. Afectarán al problema de la mezcla y la resolución de conflictos. Las líneas son:

```
Consola

prompt:~> git config --global merge.conflictstyle diff3
prompt:~> git config --global merge.tool meld
```

El uso de **diff3** le permite tener una información más completa sobre el conflicto y el programa **meld** le puede resultar cómodo para visualizar gráficamente qué partes coinciden y qué partes están en conflicto. Tal vez la más interesante sea la primera, pues completa con más información el conflicto no sólo informando de las diferencias, sino también del contenido común del que surgió.

La segunda le habilita usar la orden **mergetool**, pues al configurar el programa **meld** como herramienta gráfica para resolver conflictos le puede resultar mucho más simple recorrerlos e incluso modificarlos con un simple *click* del ratón. Al terminar la ejecución de **meld**, la orden pregunta si se resolvió el conflicto para pasar el archivo al **stage** y si sigue resolviendo. Por ejemplo, en nuestro ejemplo obtengo lo siguiente:



Por supuesto, son dos opciones que habitualmente uso y que sospecho que le pueden resultar cómodas. La configuración que finalmente le parezca mejor podría ser otra. Probablemente la primera le resultará más interesante. Si quiere, puede ignorar el programa **meld** ahora mismo, aunque puede usarlo si quiere ver gráficamente las tres versiones.

Resolviendo el conflicto

La resolución del conflicto consiste en hacer un nuevo **commit** con la versión combinada. Dicho de otra forma, tiene que modificar los archivos en conflicto, pasarlo al **stage** y confirmar una nueva versión. Puede usar la orden **status** para comprobar su estado, verá que le indica los conflictos a resolver:

```
Consola

dev1-prompt:~/cpp/mensajes> git status
En la rama master
Your branch and 'origin/master' have diverged,
and have 1 and 1 different commit each, respectively.
(use "git pull" to merge the remote branch into yours)
Tiene rutas sin fusionar.
    (solucione los conflictos y ejecute «git commit»)

Rutas no combinadas:
    (use «git add <archivo>...» para marcar resolución)

    modificado por ambos:    saludar.cpp

no hay cambios agregados al commit (use «git add» o «git commit -a»)
```

Note que habla de ramas que divergen. Si piensa en nuestro «**HEAD**» y la nueva versión que hay en el remoto, puede imaginar que las dos son caminos que se han separado.

Como el desarrollador 1 ha configurado **diff3**, tendrá que solucionar el conflicto que aparece en **saludar.cpp** con ese formato. En concreto, obtiene:

```
#include <iostream>
int main()
{
```

```
<<<<< HEAD
  std::cout << "Comenzamos la sesión." << std::endl;
|||||| merged common ancestors
  cout << "Comenzamos la sesión." << endl;
=====
  cout << "Comenzamos la sesión..." << endl;
>>>>> refs/remotes/origin/master
}
```

donde puede ver que en medio de las dos versiones se incluye lo que tienen en común, el contenido del ancestro común. En este caso no era especialmente necesario, aunque en otros casos le podrá ayudar a entender los dos caminos que han tomado ambas versiones.

En este momento no se debe preocupar mucho de ramas, aunque puede ser ilustrativo mostrar el `log` con algo más de información, incluyendo gráfico de caminos e indicando que queremos ver otras etiquetas, como sigue:

```
dev1-prompt:~/cpp/mensajes> git log --oneline --decorate --graph --all
* bdcc027 (origin/master, origin/HEAD) ... pausa que expresa suspense
| * 803187a (HEAD -> master) Evitamos abrir std
|/
* 637a813 Cmake 3.0 y C++ 14
* 5816e43 Ficheros a ignorar
* 12cd343 Formateo
* 0d6b3a7 Configuración cmake
* 4065456 Versión inicial
```

En nuestro caso, resolvemos el conflicto con un simple editor que cambia el archivo `saludar.cpp` a:

```
#include <iostream>

int main()
{
    std::cout << "Comenzamos la sesión..." << std::endl;
```

Una vez modificado, lo añadimos al `stage` con `add`. Puede consultar con `git status` que todos los conflictos están solucionados y se espera el `commit` para resolverlos:

```
dev1-prompt:~/cpp/mensajes> git commit -m "Combinar puntos y std"
[master a402abd] Combinar puntos y std
```

Ahora podemos ver el resultado gráficamente con la unión de ambos:

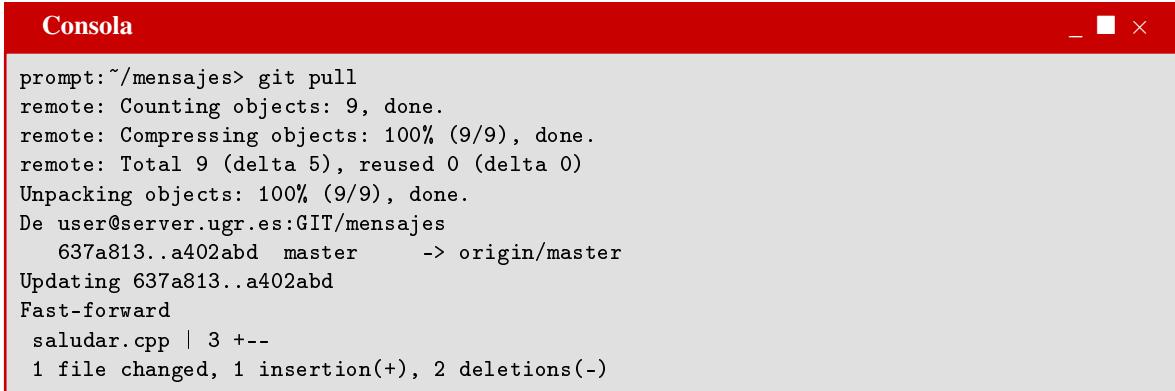
```
dev1-prompt:~/cpp/mensajes> git log --oneline --decorate --graph --all
* a402abd (HEAD -> master) Combinar puntos y std
|\ \
| * bdcc027 (origin/master, origin/HEAD) ... pausa que expresa suspense
* | 803187a Evitamos abrir std
|/
* 637a813 Cmake 3.0 y C++ 14
* 5816e43 Ficheros a ignorar
* 12cd343 Formateo
* 0d6b3a7 Configuración cmake
* 4065456 Versión inicial
```

Y terminar nuestra tarea de subir los cambios una vez fusionados:

```
dev1-prompt:~/cpp/mensajes> git push origin master
Counting objects: 6, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 625 bytes | 0 bytes/s, done.
Total 6 (delta 3), reused 0 (delta 0)
To user@server.ugr.es:GIT/mensajes.git
 bdcc027..a402abd master -> master
```

Descargar sin haber cambiado el local

Nosotros, como desarrollador inicial no hemos hecho ningún cambio, pero queremos conocer la última versión del repositorio común. Para hacerlo, tendremos que hacer **fetch** y **merge**. Sin embargo, usamos una solución más simple que en la práctica realiza las dos cosas de forma consecutiva: **pull**.



```
prompt:~/mensajes> git pull
remote: Counting objects: 9, done.
remote: Compressing objects: 100% (9/9), done.
remote: Total 9 (delta 5), reused 0 (delta 0)
Unpacking objects: 100% (9/9), done.
From user@server.ugr.es:GIT/mensajes
 637a813..a402abd master      -> origin/master
Updating 637a813..a402abd
Fast-forward
  saludar.cpp | 3 +--
  1 file changed, 1 insertion(+), 2 deletions(-)
```

Esta operación no podía fallar, no podía haber conflictos, pues la versión que tenemos ya era parte de lo que había en el remoto. Si consulta el estado, verá que tenemos todos los cambios en nuestro repositorio local

Es una operación muy habitual y que no debe temer si sólo está siguiendo el desarrollo de un proyecto. Si quiere tener la última versión, vaya al árbol con el proyecto y haga **pull**.

Con lo que terminamos este tutorial de fundamentos. Con lo que hemos explicado, ya puede empezar a trabajar con «**Git**». Quién le iba a decir que un programa tipo *Hola mundo* iba a dar para decenas de páginas de explicación...¹²

¹² “Este caso de puntos suspensivos se podría considerar la interrupción voluntaria de un discurso cuyo final se da por conocido o sobrentendido por el interlocutor”. El programador 2.

```

#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;
}

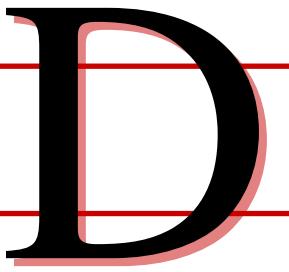
int main()
{
    Fecha f1,f2;

    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 155

Indicaciones generales 155

Énfasis automático

Reglas de estilo

Guía de estilo 156

Identificadores

Estructura del código

Consideraciones adicionales

If (is >> f) {

if (f && (c & 0)) {

is.setstate(std::ios::failbit);

return is;

D.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 figuras¹ 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 convierta en un código mejor.

D.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.

¹Puede consultar “código ofuscado” en la red para descubrir múltiples programas y resultados.

D.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.

D.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.

D.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.

D.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 D.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 D.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 D.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 D.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 D.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 eficiente² que asocia una constante global con la directiva `#define` y que por tanto se escribe en mayúsculas.

Regla D.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 D.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 D.8 — Tipos globales. Los identificadores de tipos definidos por el usuario usarán las mayúsculas, incluyendo la primera de las letras (*Pascal Case*). Si lo desea distinguir las funciones, puede usar el carácter `'_'` para separar las palabras del identificador.

Regla D.9 — Funciones globales. Los identificadores de funciones globales usarán las mayúsculas sólo para la primera de las letras. El resto será en minúscula, incluyendo el carácter `'_'` si es necesario.

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 D.10 — 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*³. 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 desaconsejan⁴.

²Aunque también se pueden considerar otros inconvenientes o desventajas frente al uso de un objeto definido con `const`.

³El origen del nombre se debe al creador de la notación, *Charles Simonyi*, con relación a su origen —nació en Hungría— y su lengua natal.

⁴Incluso *Microsoft* en algunas de sus especificaciones llega a indicar explícitamente que no se debe usar.

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.
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 caso⁵. 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 D.11 — 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;
    // ...
};
```

Otra posibilidad es evitar este carácter como prefijo añadiéndolo como posfijo. Así, en el ejemplo anterior, podría usarse:

```
class Fecha {
    int dia_;
    // ...
};
```

Finalmente, es interesante añadir un último comentario sobre los *posfijos*. En la práctica, pueden pasar más fácilmente desapercibidos⁶, por lo que podríamos considerar evitarlos; esto es especialmente importante si tenemos varios identificadores similares que se diferencian sólo en la parte final.

A pesar de lo anterior, 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 para el precompilador, se escribe enteramente en mayúscula y probablemente con un primer carácter '*_*' que enfatiza su carácter interno o reservado.

D.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

⁵Aun 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.

⁶También depende del entorno de desarrollo, que en muchos casos usa un tipo de letra y color que facilita su identificación.

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.

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 tabulador⁷. 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 línea⁸. En un programa se debe establecer un número fijo de espacios por cada nivel de sangrado.

Regla D.12 — 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 D.13 — 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 D.14 — Líneas largas. Se debe evitar dejar que el editor sea el responsable de formatear las líneas largas. Debemos ser nosotros los que formateeemos 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:

⁷Tenga 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.

⁸Es 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*”.

```
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.

Regla D.15 — 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 D.16 — 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 D.17 — 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) {
    // Suspensión
}
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 D.18 — 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 D.19 — 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 D.20 — 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 D.21 — 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 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 D.22 — 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 D.23 — 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 D.24 — 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.

D.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 globales⁹.

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 almacenado¹⁰, 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 D.25 — 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 D.26 — 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`.

⁹De 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.

¹⁰No 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.

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.

Regla D.27 — 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 D.28 — 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 D.29 — 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 D.30 — 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`, `.hxx`, `.hh`, `.cpp`, `.cxx`, `.c++`, `.tcc`, etc.

Regla D.31 — 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. En principio, adelantamos la siguiente regla:

Regla D.32 — Nombre de archivos. Definiciones de plantillas. Los nombres de archivos plantilla con definiciones que se incluyen en los archivos cabecera `.h` para que el compilador disponga de las plantillas para instanciar tendrán extensión `.hxx`.

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 D.33 — 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 D.34 — 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 D.35 — 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.



Generación de números aleatorios

F

Introducción	171
El problema	171
Números pseudoaleatorios	
Transformación del intervalo	173
Operación módulo	
Normalizar a U(0,1)	

E.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.

E.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.

E.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 usarán 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.

E.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.

E.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.

E.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;
}
```


F

Tablas

Tabla ASCII	175
Operadores C++	176
Palabras reservadas de C89, C99, C11, C++ y C++11	177
Manipuladores y funciones miembro para E/S formateada	177
Banderas y máscaras	
Funciones miembro y manipuladores	
Referencia de operaciones con GIT	179

F.1 Tabla ASCII

En la figura F.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	¿
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 F.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.

F.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	espacio_nombres::miembro
::	Resolución de ámbito	<i>nombre_clase</i> ::miembro
->	selección de miembro	<i>puntero</i> ->miembro
.	Selección de miembro	<i>objeto</i> .miembro
[]	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> --
typeid	Identificador de tipo	typeid(<i>type</i>)
typeid	... en tiempo de ejecución	typeid(<i>expr</i>)
dynamic_cast	conversión en ejecución	dynamic_cast< <i>tipo</i> >(<i>expr</i>)
	con verificación	
static_cast	conversión en compilación	static_cast< <i>tipo</i> >(<i>expr</i>)
	con verificación	
reinterpret_cast	conversión sin verificación	reinterpret_cast< <i>tipo</i> >(<i>expr</i>)
const_cast	conversión const	const_cast< <i>tipo</i> >(<i>expr</i>)
sizeof	Tamaño del tipo	sizeof(<i>tipo</i>)
sizeof	Tamaño del objeto	sizeof <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>
new	reserva	new <i>tipo</i>
new	reserva e iniciación	new <i>tipo</i> (<i>lista_expr</i>)

continúa en la página siguiente

continúa de la página anterior		
Operador	Nombre	Uso
<code>new</code>	reserva (emplazamiento)	<code>new (lista_expr) tipo</code>
<code>new</code>	reserva (con inicialización)	<code>new (lista_expr) tipo(lista_expr)</code>
<code>delete</code>	destrucción (liberación)	<code>delete puntero</code>
<code>delete []</code>	... de un vector	<code>delete [] puntero</code>
<code>()</code>	Conversión de tipo	<code>(tipo) expr</code>
<code>. *</code>	Selección de miembro	<code>objeto.*puntero_a_miembro</code>
<code>->*</code>	Selección de miembro	<code>puntero->*puntero_a_miembro</code>
<code>*</code>	Multiplicación	<code>expr*expr</code>
<code>/</code>	División	<code>expr/expr</code>
<code>%</code>	Módulo	<code>expr%expr</code>
<code>+</code>	Suma	<code>expr+expr</code>
<code>-</code>	Resta	<code>expr-expr</code>
<code><<</code>	Desplazamiento a izquierda	<code>expr<<expr</code>
<code>>></code>	Desplazamiento a derecha	<code>expr>>expr</code>
<code><</code>	Menor	<code>expr<expr</code>
<code><=</code>	Menor o igual	<code>expr<=expr</code>
<code>></code>	Mayor	<code>expr>expr</code>
<code>>=</code>	Mayor o igual	<code>expr>=expr</code>
<code>==</code>	Igual	<code>expr==expr</code>
<code>!=</code>	No igual	<code>expr!=expr</code>
<code>&</code>	Y a nivel de bit	<code>expr&expr</code>
<code>^</code>	O exclusivo a nivel de bit	<code>expr^expr</code>
<code> </code>	O a nivel de bit	<code>expr expr</code>
<code>&&</code>	Y lógico	<code>expr&&expr</code>
<code> </code>	O lógico	<code>expr expr</code>
<code>? :</code>	expresión condicional	<code>expr?expr:expr</code>
<code>=</code>	Asignación	<code>valor_i=expr</code>
<code>*=</code>	Multiplicación y asignación	<code>valor_i*=expr</code>
<code>/=</code>	División y asignación	<code>valor_i/=expr</code>
<code>%=</code>	Resto y asignación	<code>valor_i%=expr</code>
<code>+=</code>	Suma y asignación	<code>valor_i+=expr</code>
<code>-=</code>	Resta y asignación	<code>valor_i-=expr</code>
<code><<=</code>	Desplazar izq. y asignación	<code>valor_i<<=expr</code>
<code>>>=</code>	Desplazar der. y asignación	<code>valor_i>>=expr</code>
<code>&=</code>	Y y asignación	<code>valor_i&=expr</code>
<code>^=</code>	O exclusivo y asignación	<code>valor_i^=expr</code>
<code> =</code>	O y asignación	<code>valor_i =expr</code>
<code>throw</code>	Lanzamiento de excepción	<code>throw expr</code>
<code>,</code>	Coma	<code>expr ,expr</code>

Tabla F.1
Operadores de C++

Estos operadores están incluidos en el estándar C++98. En C++11 aparecen tres más:

`sizeof..., noexcept, alignof`

F.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 F.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.

F.4 Manipuladores y funciones miembro para E/S formateada

En la siguiente figura F.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:

Tabla F.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	

- 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.

F.4.1 Banderas y máscaras

Si observa la figura F.2 (página 180) 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— podemos 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 F.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.

F.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 F.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.

F.5 Referencia de operaciones con GIT

En las tablas F.3 y F.4 se muestran gráficamente las operaciones del tema C. Corresponden a las figuras reunidas en dos páginas para que sirvan como referencia.

	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)		
	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			
	copyfmt(str)	Copia las banderas desde flujo str			
	Ancho de campo, relleno y ajuste (números, bool, cadena C, etc.)				
	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)		
	máscara adjustfield				
	left	Ajusta a la izquierda	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)		
	Números enteros	Números			
	showpos	Escribe el signo a los números positivos	⟨ showpos no showpos		
	uppercase	Usa mayúsculas para números (hexadecimal y científica)	⟨ uppercase no uppercase		
	máscara basefield				
	oct	Escribe y lee octal	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)		
	showbase	Escribe 0 para octal y 0x para hexadecimal	⟨ showbase no showbase		
	máscara floatfield				
	fixed	Notación fija (precisión indica el número de dígitos después de la coma)	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)			
	precision(p)	Selecciona precisión p y devuelve anterior	setprecision(p)		
	showpoint	Escribe ceros a la derecha del punto decimal			
	bool	boolalpha	Si los bool se manejan textualmente (defecto 0/1). El texto depende de idioma	⟨ boolalpha no boolalpha	
En istream y ostream	flush()	Descarga búfer de salida Inserta salto de línea y descarga búfer Inserta carácter fin de cadena Lee e ignora "espacios blancos"	flush endl ends ws	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 de una bandera en una máscara desactivan el resto de ésta	
Otros	skipws	Salta "espacios blancos" iniciales en cada lectura (por defecto)	⟨ skipws no skipws		
	unitbuf	Descarga búfer de salida tras cada escritura (por defecto para cerr y wcerr)	⟨ unitbuf no unitbuf		

Figura F.2
CheatSheet para E/S formateada.

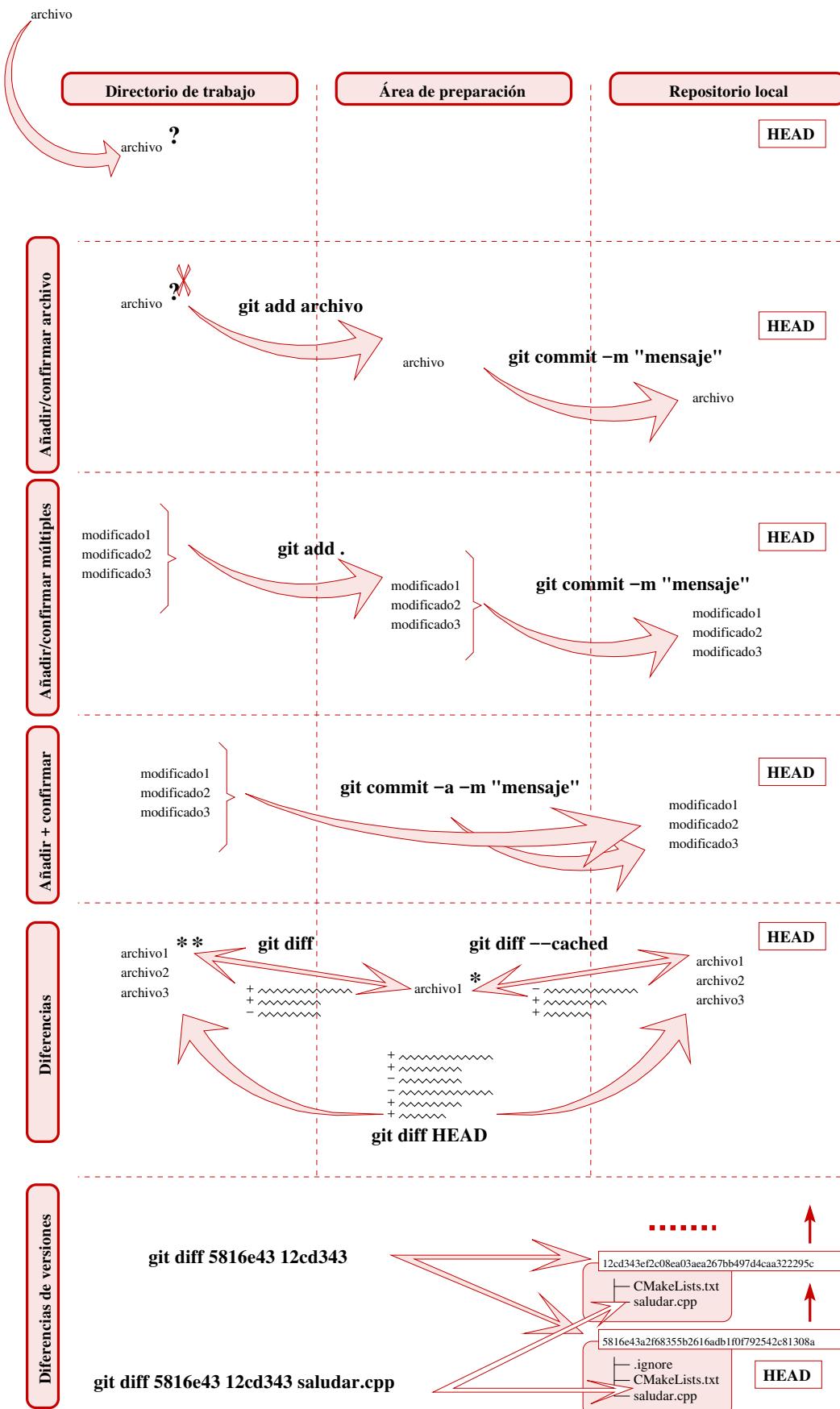


Figura F.3
CheatSheet para Git (página 1).

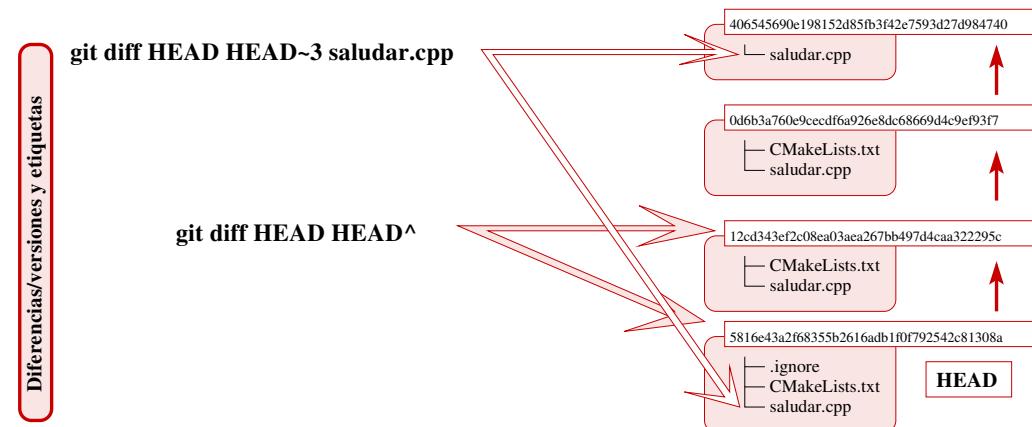
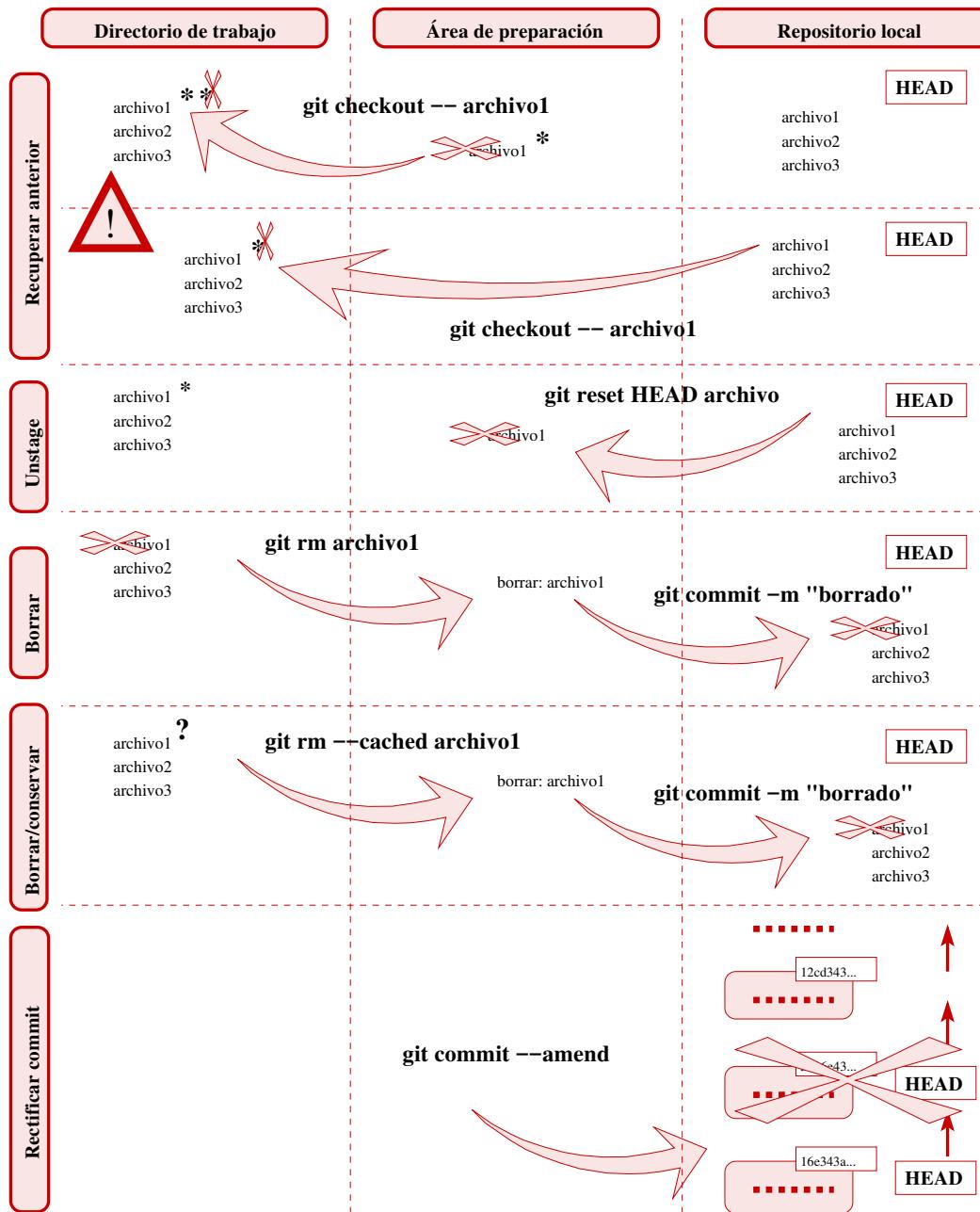


Figura F.4
CheatSheet para Git (página 2).

Bibliografía

Referencias principales

- [1] Garrido, A. *Fundamentos de programación con la STL*. Editorial Universidad de Granada, 2016.
- [2] Garrido, A. *Metodología de la Programación: de bits a objetos*. Editorial Universidad de Granada, 2016.

Referencias secundarias

- [3] Scott Chacon y Ben Straub, *Pro Git*. Segunda Edición. Apress. 2014. Disponible también como referencia electrónica.
- [4] Deitel y Deitel, *C++11 for programmers* . Segunda Edición. Prentice Hall. 2013.
- [5] Gonzalez, R. y Wood, R. *Digital Image Processing*. Prentice Hall, 2002.
- [6] Garrido, A. *Fundamentos de programación en C++*. Delta publicaciones, 2005.
- [7] Garrido, A, Fernández Valdivia, J. *Abstracción y estructuras de datos en C++*. Delta publicaciones, 2005.
- [8] Donald E. Knuth, *The Art of Computer Programming*, vol. 3 (Sorting and Searching). Addison-Wesley Publishing Company, 2nd edition, 1998.
- [9] Sedgewick, *Algorithms in C++*, Addison Wesley, 1998.
- [10] Stroustrup, B. *El lenguaje de programación C++*. Edición Especial. Addison-Wesley, 2002.
- [11] Stroustrup, B. *The C++ Programming Language, 4th Edition*. Addison-Wesley, 2013.
- [12] Stroustrup, B. *The design and Evolution of C++*. Addison-Wesley, 1994.
- [13] Stroustrup, B. *Programming: Principles and Practice Using C++*. Segunda edición. Addison-Wesley, 2014.

Referencias electrónicas

- [14] Scott Chacon y Ben Straub, *Pro Git*. Segunda Edición. Apress. 2014. <https://git-scm.com/book/es/v2>
- [15] *C++ Reference*. Página web con una referencia bastante completa de los recursos disponibles en el lenguaje C++. <http://www.cplusplus.com/reference/>
- [16] *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/>
- [17] B. Stroustrup, *C++ Style and Technique FAQ*. http://www.research.att.com/bs/bs_faq2.html.
- [18] GNU, *GNU Make*. <http://www.gnu.org/software/make/manual/make.html>.

Índice alfabético



Símbolos

.gitignore, 132
QtCreator y, 113

A

acceso aleatorio
celdas enlazadas y, 68
alias
git y, 142
and
a nivel de bit, 28
matriz booleana, 36
ar, 14
archivos cabecera, estilo, 168
áreas de almacenamiento, 125
argc, 24
argv, 24
aritmético
tipo, 1
ASCII, 175
atoi, 46, 50

B

bandera
E/S y, 178
biblioteca, 13
orden de las, 15
bidireccional, burbuja, 55
binaria interpolada, búsqueda, 54
binaria, búsqueda, 52
bit
operador a nivel de, 28
bool, 4
matriz de, 35
bucle
vacío, estilo, 164
burbuja, ordenación por, 54
búsqueda, 50
binaria con interpolación, 54
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, 158
casting, 5
cat, 36
cdash, 90
celdas enlazadas, 64
mezcla, 69
ordenación, 68
puntero a puntero a celda, 67
rango de, 71
cfloat, 2
char, 1
chunk
git diff y, 138
class, estilo, 165
climits, 2
cmake, 89, 110
biblioteca con, 97
booleanos, 101
condicionales, 102
función, 104

git y merge, 151
constante
estilo de, 159
control de versiones, 123
conversión
explícita, 5
implícita, 4
cpack, 90
ctest, 90, 105
opciones, 107
CXX, 18
CXXFLAGS, 18

D

dec, 3
declaración, estilo, 167
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, 158
do-while, estilo, 164
double, 1
doxygen, 32

E

E/S
redirecciónamiento 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, 166
enum, estilo, 166
espacio blanco
estilo y, 161
esteganografía, 32
estilo
archivos cabecera, 168

bucle vacío, 164
 class, 165
 comentario, 168
 compilación separada, 168
 declaración, 167
 do-while, 164
 enum, 166
 enum class, 166
 for, 164
 función global, 165
 if-else, 162
 private/protected/public, 167
 punteros y referencias, 167
 reglas de, 156
 struct, 165
 struct vs class, 166
 switch, 163
 while, 164

F

float, 1
 for, estilo, 164
 función
 global, estilo, 159, 165
 miembro, estilo de, 159
 función de biblioteca
 atoi, 46, 50
 rand, 46, 50, 171
 srand, 46, 50, 172
 strcmp, 49
 time, 46, 50, 172
 función de cmake, 104

G

g++, 10
 git, 123
 .gitignore, 132
 QtCreator y, 113
 add, 126
 add ., 129
 área de preparación, 126
 áreas de almacenamiento, 125, 133
 áreas de almacenamiento vs versiones, 135
 branch, 148
 checkout, 136
 clone
 bare, 146
 url, 147
 commit, 126
 - -amend, 144
 commit -a, 130
 mensajes, edición, 132
 rectificar último, 144
 config, 131
 alias, 142
 editor, 132
 identidad, 125, 131
 ignorar archivos, 131
 mezcla, 152
 conflicto en merge, 151
 descargar cambios, 150
 descargar y mezclar, 154
 diferencias, 133
 versiones, 143
 diff, 138
 HEAD, 140
 eliminar archivo, 131
 del repositorio, 135
 etiquetas
 ancestros, 143
 fetch, 150
 hash, 124
 HEAD, 129, 133
 historial del repositorio, 141
 index, 126, 133

índice, 126
 init
 - -bare, 146
 log, 141
 merge, 151
 diff3, 152
 mergetool, 152
 mezclar cambios, 151
 modificado (archivo), 129
 mv, 131
 preparado (archivo), 128
 protocolos del servidor, 146
 pull, 154
 push, 149
 rama
 seguir una, 147
 rama master, 127, 141, 147
 rango de revisiones, 142
 rectificar commit, 144
 remote, 147
 add, 148
 renombrar archivo, 131
 repositorio, 125
 origin, 147
 remoto, 145
 reset, 135, 144
 rm, 131
 -cached, 135
 seguimiento
 de rama, 147
 sin seguimiento (archivo), 128, 136
 staging area, 126, 133
 status, 127
 subir cambios, 149
 versión
 concepto, 124
 volver a, 144

gris
 imagen de, 30

H

hash, valor, 124
 HEAD, de git, 129, 133
 hex, 3
 húngara, notación, 160

I

IDE, 109, 156
 identificador
 estilo de, 157
 IEEE-754/IEC-559, 3
 if
 cmake, 102
 if-else, estilo, 162
 ifndef/endif, 12
 ifstream, 23
 ignorar
 archivos en git, 131
 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, 54
 iomanip, 179
 ISO8859, 5, 6, 175
 istream, 23

J

juego
 otelo, 73
 reversi, 73

K

kit, de QtCreator, 113

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, 157

M

módulo de cmake, 104
 macro
 estilo de, 159
 makefile y, 18
 macro de cmake, 104
 main, 24
 make, 15, 110
 makefile, 15, 16
 manipulador de E/S, 179
 máscara
 E/S y, 178
 master
 rama git, 127, 141, 147
 matriz
 booleana, 35
 md5sum, 124
 meld, diff3 y, 152
 mergesort, 70
 metagenerador, 110
 metagenerador, cmake como, 90
 método
 estilo de, 159
 mezcla
 de elementos ordenados, 61, 69
 ordenación por, 70
 modificado, archivo git, 129

N

NaN, 3
 new[], 46
 not
 a nivel de bit, 28
 matriz booleana, 36
 notación
 húngara, 160
 numeric_limits, 2
 números aleatorios, 171
 pseudoaleatorios, 172

O

objetivo
 regla make y, 17
 objeto
 archivos, 13
 oct, 3
 operador
 nivel de bit, 28
 operadores C++, 176
 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
 origin
 repositorio git, 147
 otelo, juego, 73

P

palabras reservadas, 177
 pascal case, 158
 PGM, formato de imagen, 31
 PHONY, 21
 píxel, 29
 posijo
 identificador con, 159
 PPM, formato de imagen, 31
 precompilador, 12
 prefijo
 identificador con, 159
 preparado, archivo git, 128
 private/protected/public, estilo, 167
 pseudoaleatorio, 172
 puntero
 a función, 56
 puntero nulo
 listas y, 64
 punteros y referencias, estilo, 167
 punto flotante, 2

Q

qmake, 110
 proyecto QtCreator, 111
 Qt, bibliotecas, 110
 QtCreator, 110

R

rand, 46, 50, 171

rango
 ordenación de un, 60
 secuencia y, 59, 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
 repositorio
 git e historial, 141
 local de git, 125
 remoto de git, 145
 reservadas
 palabras, 177
 reversi, juego, 73
 RGB, 30
 right, 3

S

sangrado, 161
 secuencial
 búsqueda, 51
 garantizada, 52
 búsqueda en rango, 59
 selección, ordenación por, 49, 69
 semilla, 172
 servidor
 de git, 146
 setfill, 3
 setw, 3
 sha1sum, 124
 shellsort, 57
 short int, 1
 signed, 1
 signed char, 5
 sin seguimiento, archivo git, 128
 sizeof, 1
 sobrecarga
 de funciones, 23
 srand, 46, 50, 172
 stage
 de git, 135

staged, de git, 128
 staging area, 126, 133
 strcmp, 49
 struct vs class, estilo, 166
 struct, estilo, 165
 switch, estilo, 163

T

tar, 20
 tests con cmake, 105
 time, 46, 50, 172
 tipo de dato
 estilo de, 159
 transformación de un rango, 60

U

unidad de compilación, 12
 union, 7
 unsigned, 1
 unsigned char, 5
 untracked, de git, 128
 UTF-8, 6, 176

V

variable
 local
 estilo de, 159
 vector
 dinámico, 46
 reservados vs usados, 47
 versión
 de git, 124

W

while, estilo, 164
 Windows-1252, 176

X

xor
 a nivel de bit, 28